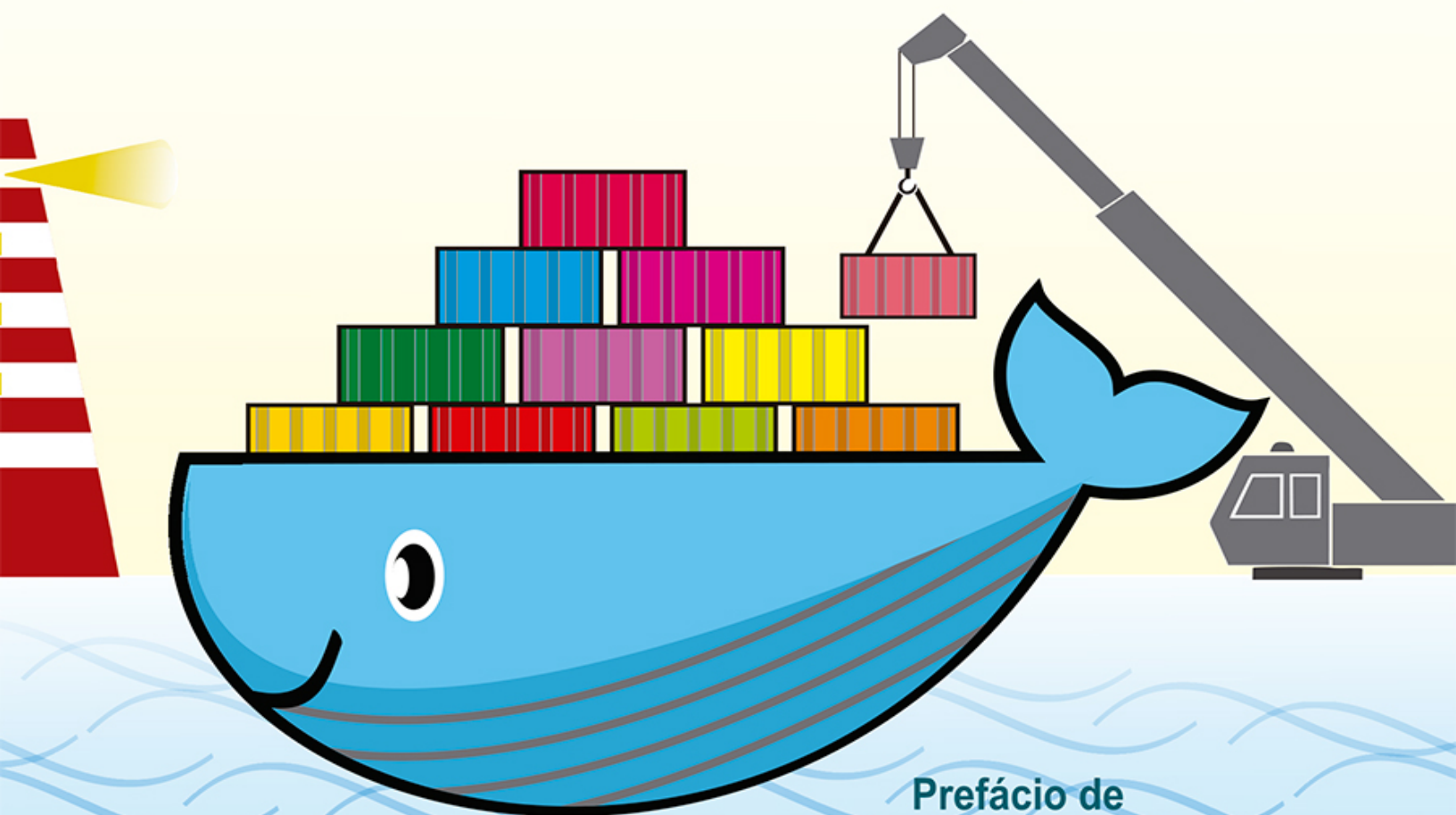


Jeferson Fernando Noronha Vitalino
Marcus André Nunes Castro

Descomplicando o Docker



Prefácio de
Jérôme Petazzoni





Descomplicando o

Docker

Jeferson Fernando Noronha Vitalino
Marcus André Nunes Castro

Descomplicando o Docker



Copyright© 2016 por Brasport Livros e Multimídia Ltda.

Todos os direitos reservados. Nenhuma parte deste livro poderá ser reproduzida, sob qualquer meio, especialmente em fotocópia (xerox), sem a permissão, por escrito, da Editora.

Para uma melhor visualização deste e-book sugerimos que mantenha seu software constantemente atualizado.

Editor: Sergio Martins de Oliveira

Diretora Editorial: Rosa Maria Oliveira de Queiroz

Gerente de Produção Editorial: Marina dos Anjos Martins de Oliveira

Copidesque: Camila Britto da Silva

Editoração Eletrônica: SBNigri Artes e Textos Ltda.

Capa: Use Design

Produção de e-pub: SBNigri Artes e Textos Ltda.

Técnica e muita atenção foram empregadas na produção deste livro. Porém, erros de digitação e/ou impressão podem ocorrer. Qualquer dúvida, inclusive de conceito, solicitamos enviar mensagem para brasport@brasport.com.br, para que nossa equipe, juntamente com o autor, possa esclarecer. A Brasport e o(s) autor(es) não assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso deste livro.

ISBN Digital: 978-85-7452-800-7

BRASPORT Livros e Multimídia Ltda.

Rua Pardal Mallet, 23 – Tijuca

20270-280 Rio de Janeiro-RJ

Tels. Fax: (21) 2568.1415/2568.1507

e-mails:

marketing@brasport.com.br

vendas@brasport.com.br

editorial@brasport.com.br

site: www.brasport.com.br

Filial

Av. Paulista, 807 – conj. 915
01311-100 – São Paulo-SP

Agradecimentos

Jeferson Fernando

Ao meu companheiro de longas jornadas de trampo, Marcus André, pessoa fundamental para o desenvolvimento deste livro. Além de ser um superamigo, me orgulha muito acompanhar seu crescimento profissional e pessoal! *Iei!!!!*

Aos alunos que acumulei e com quem fiz amizades ao longo desses 12 anos ministrando treinamentos pelo Brasil, especialmente aos alunos do treinamento *Descomplicando o Docker* da LINUXtips! :D

A todos os amigos que fiz por onde passei, especialmente o Juliano “ncode” Martinez, João “orelhinhas” Gabriel, Jean Feltrin, Guilherme “Lero” Schroeder, Rodolfo “enetepe” Ponteado, Guilherme Almeida, Luiz Salvador Jr, Jhonatan Araujo, Francisco “chico” Freire, Adinan Paiva, Adilson Nunes, Pedro Vara, Eduardo Lipolis, Fábio “santa” Santiago, Giordano Fechio, Rafael Benvenuti, João Borges, Daniel Lima, Mateus Prado e Leo Martins.

Com muito carinho, gostaria de agradecer a duas pessoas que foram muito importantes para mim nesses últimos anos, principalmente em relação a minha carreira: Eduardo Scarpellini e Leonardo Lorieri, muito obrigado! Como sempre digo, vocês mudaram a minha vida! <3

Meu chefes:

Rogério Lelis, que sempre me ajudou e me apoiou em tudo que precisei! André Brandão, que, além de ser o melhor gerente que já tive, é o meu *coach* e me ajudou a quebrar algumas barreiras bem importantes em minha vida.

Rodrigo Campos, que tenho como espelho pela sensacional habilidade em palestrar com maestria.

Quero agradecer muito aos meus irmãos Magno Junior e Camila Vitalino, sempre ao meu lado e me apoiando em tudo. Amo vocês!

O agradecimento mais especial é para as mulheres da minha vida:

⇒ Minha esposa e fiel companheira Aletheia, que sempre está ao meu lado em todos os momentos, tristes ou felizes, fáceis ou difíceis, e que sempre acreditou em mim. Eu te amo!

⇒ Minhas princesas e as coisas mais lindas, Maria Fernanda e Maria Eduarda, que são meu tudo, meu ar, minha vida! Amo vocês, princesas!

⇒ E finalmente minha rainha, minha mãe, Cidinha Tavares, que passou por diversas dificuldades para conseguir educar o homem que hoje sou! Te amo!

Marcus André

Eu agradeço, primeiramente, ao Jeferson Vitalino (quem?) pela ideia, oportunidade e parceria neste livro. Gostaria, também, de continuar agradecendo a ele pelo tempo, pela dedicação e, acima de tudo, por acreditar em mim. Esses últimos anos têm sido, sem dúvida, os melhores da minha carreira profissional e você tem uma bela parcela de culpa nisso.

Queria também agradecer a todas as pessoas que contribuíram direta ou indiretamente para a minha formação como profissional de TI: colegas de trabalho, professores da faculdade, chefes, ex-clientes, pessoas que me agregaram muito valor e às quais eu serei eternamente grato.

Um agradecimento especial aos meus amigos/irmãos: Mário Anderson, Márcio Ângelo, Bruno Henrique e Manuela Guedes. Meus amigos Juciano Werllen, Rondineli Gomes, Renan Vicente e amigos de outras áreas: Jémina Diógenes, Juliana Mesquita, Ana Clara Rezende e Rodrigo Peixe.

À minha família: meu pai, José Monte, e minha mãe, dona Evani, muito obrigado por tudo e parabéns, seu projeto de homem feliz deu certo. :)

À minha mais nova família, Elisângela Oliveira. Agradeço imensamente pela compreensão nesses últimos meses, pelo carinho, pela emoção, pelo cuidado. Muito obrigado por me mostrar a complexidade nas coisas simples e o universo que existe entre o **0** e o **1**. Eu te amo.

Prefácio

If you are a developer, you probably have heard about Docker by now: Docker is the ideal platform to run applications in containers. Fine, but what are those containers? Containers are a lightweight virtualization technique providing us with lots of possibilities: thanks to them, we can ship applications faster; we can easily implement CI/CD (continuous integration and continuous delivery); we can set up development environments faster than ever; we can ensure parity between those development environments and our production servers; and much more. If you write code, if you deploy code, if you operate code, then I promise that containers are going to make your life easier.

But containers have been around for *decades*, and Docker was only released in 2013. So what's the big deal? Why is everybody so excited about Docker, if the technology behind it is more than ten years old?

Because the main innovation of Docker is not the technology. The main innovation of Docker is to make this technology available to every developer and sysadmin, very easily, without having to spend years of practice to become an expert, and without requiring to develop tons of custom tools.

I will share a secret with you: the really important thing is not Docker and containers. What is really important is to improve the quality of our software, reduce our development and deployment costs, release better, more reliable code, faster. It turns out that Docker is an insanely efficient way to achieve those goals. That's why it is so popular.

This book will teach you all you need to know to get started with Docker, and use it to build, ship, and run your applications. It will be your guide in the world of containers, and on the path to ship code better than you ever did before.

Jérôme Petazzoni

Docker Tinkerer Extraordinaire

Sobre o Livro

A proposta deste livro é ajudá-lo a construir ambientes utilizando *containers* com a ferramenta que está revolucionando as empresas de tecnologia ao redor do mundo: vamos aprender e brincar bastante com o sensacional Docker!

De maneira leve e totalmente prática, vamos aprender desde o que é o Docker até a criação de um *cluster* Docker com diversos *containers* em execução! Sempre de forma prática, vamos abordar temas importantes para que consiga administrar ambientes que utilizam ou pretendem utilizar o Docker.

Inclusive, vamos aprender a montar *dockerfiles* personalizados para construção de imagens de *containers*, além de conhecer como melhor administrá-las.

Também veremos na prática como utilizar o Docker Machine para criação de *hosts* Docker, seja local ou na nuvem. Vamos criar um *cluster* utilizando o Swarm e aprender como escalar o nosso ambiente através do Compose.

Sobre os Autores

Jeferson Fernando

Profissional com mais de 14 anos de experiência em administração de servidores Unix/Linux em ambientes críticos de grandes empresas como Motorola, TIVIT, Votorantim, Alog, Locaweb e Walmart.com. Possuindo mais de 12 anos de experiência como instrutor em grandes centros de treinamento, passou por Impacta Tecnologia, Utah Linux Center, Green, 4Linux e foi instrutor oficial da Red Hat. Já ministrou treinamentos como consultor em empresas como Petrobras, Vale, Ministério do Exército, Polícia Militar do Estado de SP e HP. Possui diversas certificações, como LPI, RHCE, RHCI, ITIL, Solaris, entre outras. É um grande entusiasta do software livre e especialista na administração de ambientes críticos com cultura DevOps. Possui ainda um canal no YouTube chamado LinuxTips, onde aborda temas interessantes para todos os níveis de administradores de sistemas Linux.

Marcus André

Profissional com oito anos de experiência em administração de redes e sistemas em pequenas/médias/grandes empresas, tendo trabalhado em diversas plataformas e ambiente homogêneo (*nix, Windows).

Focado em FOSS (*Free and Open Source Software*), atualmente integra o time de *Systems Administrators* responsável pelo acompanhamento e pela administração da infraestrutura de múltiplos *e-commerces* do grupo Walmart

na América Latina. Evangelista da linguagem Python – apesar de não se considerar programador – e apaixonado por toda a cultura DevOps.

Sumário

Introdução

1. O que é container?

1.1. Então vamos lá, o que é um container?

1.2. E quando começou que eu não vi?

2. O que é o Docker?

2.1. Onde entra o Docker nessa história?

2.2. E esse negócio de camadas?

2.2.1. Copy-On-Write (COW) e Docker

2.3. Storage drivers

2.3.1. AUFS (Another Union File System).

2.3.2. Device Mapper

2.4. Docker Internals

2.5. Namespaces

2.5.1. PID namespace

[2.5.2. Net namespace](#)

[2.5.3. IPC namespace](#)

[2.5.4. Mnt namespace](#)

[2.5.5. UTS namespace](#)

[2.5.6. User namespace](#)

[2.6. Cgroups](#)

[2.7. Netfilter](#)

[2.8. Para quem ele é bom?](#)

[3. Instalando o Docker](#)

[3.1. Quero instalar, vamos lá?](#)

[3.1.1. Dica importante](#)

[3.2. Windows, MacOS, etc.](#)

[4. Executando e administrando containers Docker](#)

[4.1. Então vamos brincar com esse tal de container!](#)

[4.2. Legal, quero mais!](#)

[4.2.1. Modo interativo](#)

[4.2.2. Daemonizando o container](#)

[4.3. Entendi, agora vamos praticar um pouco?](#)

[4.4. Tá, agora quero sair...](#)

[4.5. Posso voltar ao container?](#)

[4.6. Continuando com a brincadeira...](#)

[4.7. Subindo e baixando containers...](#)

[4.8. Visualizando o consumo de recursos pelo container](#)

[4.9. Cansei de brincar de container, quero removê-lo!](#)

[5. Configurando CPU e memória para os meus containers](#)

[5.1. Especificando a quantidade de memória](#)

[5.2. Especificando a quantidade de CPU](#)

[5.3. Eu consigo alterar CPU e memória dos meus containers em execução?](#)

[6. Meu primeiro e tosco dockerfile...](#)

[6.1. Outras opções](#)

[7. Entendendo volumes](#)

[7.1. Introdução a volumes no Docker](#)

[7.2. Localizando volumes](#)

[7.3. Criando e montando um data-only container](#)

[7.4. Sempre é bom ter um backup](#)

[8. Criando e gerenciando imagens](#)

[8.1. Agora eu quero criar minha imagem, posso?](#)

[8.2. Vamos começar do começo então, dockerfile!](#)

[8.3. Vamos customizar uma imagem base agora?](#)

9. Compartilhando as imagens

9.1. O que é o Docker Hub?

9.2. Vamos criar uma conta?

9.3. Agora vamos compartilhar essas imagens na interwebs!

9.4. Não confio na internet; posso criar o meu registry local?

10. Gerenciando a rede dos containers

10.1. Consigo fazer com que a porta do container responda na porta do host?

10.2. E como ele faz isso? Mágica?

11. Controlando o daemon do Docker

11.1. O Docker sempre utiliza 172.16.X.X ou posso configurar outro intervalo de IP?

11.2. Opções de sockets

11.2.1. Unix Domain Socket

11.2.2. TCP

11.3. Opções de storage

11.4. Opções diversas

12. Utilizando Docker Machine, Docker Swarm e Docker Compose

12.1. Ouvi dizer que minha vida ficaria melhor com o Docker Machine!

12.1.1. Vamos instalar?

12.1.2. Vamos iniciar nosso primeiro projeto?

12.2. Agora vamos brincar com o Docker Swarm?

12.2.1. Criando o nosso cluster!

12.2.2. Docker Swarm Discovery

12.2.3. Mas para que isso serve?

12.3. E o tal do Docker Compose?

12.3.1. Compose File

12.3.2. Instalando o Docker Compose

12.3.3. Parâmetros do Compose File

12.3.4. Vamos fazer o nosso projeto?

12.4. E já acabou? :(

Introdução

Se você é um desenvolvedor que sempre sonhou em poder reproduzir o ambiente de produção em sua máquina ou a aplicação que você desenhou e desenvolveu poder rodar em produção facilmente, com o mesmo “binário” que foi utilizado nos momentos de testes...

...Imagine conseguir instalar e testar diversas aplicações rapidamente e ainda simular um ambiente completo, bem mais leve e inteligente do que as máquinas virtuais em questão de minutos.

Se você é um administrador de sistemas que sempre sonhou em não precisar mais preparar os servidores para suportar milhares de aplicações diferentes, feitas em linguagens diferentes, com bibliotecas diferentes, com versões diferentes (ufa!)...

...Imagine economizar milhões de horas em instalações de novos servidores, atualizações de aplicações, *deploys*, acionamentos na madrugada por conta de serviços que “falharam” e não poderiam ser executados em outros servidores por incompatibilidades diversas.

Se você precisa escalar seu ambiente em questão de minutos para suportar uma demanda que o pegou de surpresa após uma campanha do departamento de marketing de sua empresa...

Se você é um gestor e está preocupado com a fatura do *datacenter* no final do mês por conta da quantidade de recursos necessários para seu ambiente funcionar....

E mesmo quando você possui todos os recursos computacionais para suportar a demanda, se você percebe que na maioria das vezes os seus recursos estão ociosos e sua equipe está com dificuldades para equalizá-los...

...Então este livro e o Docker foram feitos para você! Boa leitura e divirta-se!

1. O que é *container*?

1.1. Então vamos lá, o que é um *container*?

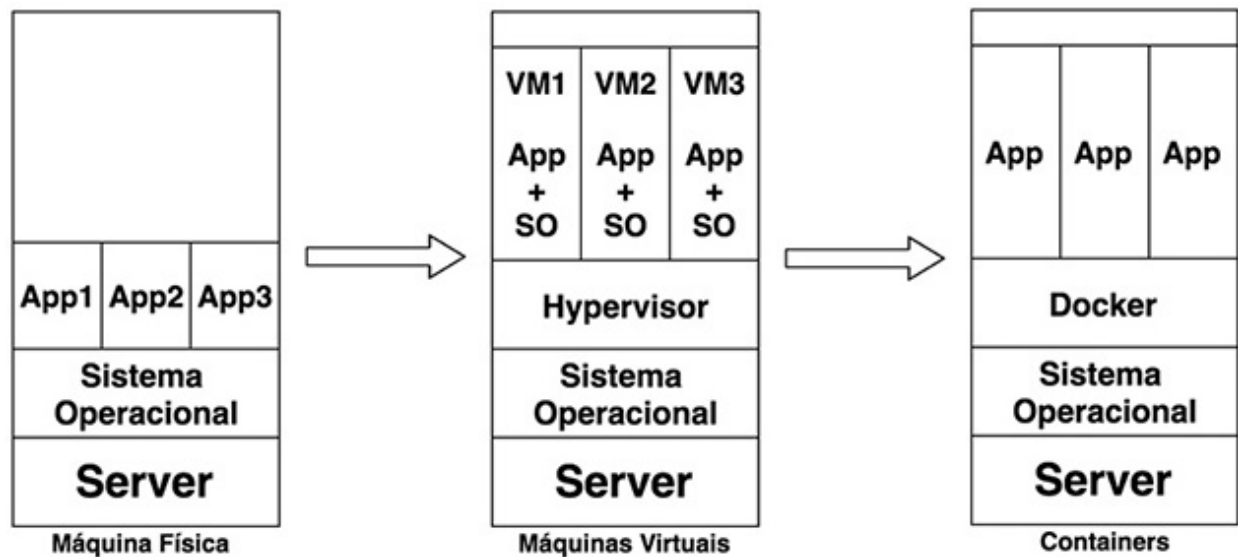
Container é, em português claro, o agrupamento de uma aplicação junto com suas dependências, que compartilham o *kernel* do sistema operacional do *host*, ou seja, da máquina (seja ela virtual ou física) onde está rodando. Deu para entender?

Containers são bem similares às máquinas virtuais, porém mais leves e mais integrados ao sistema operacional da máquina *host*, uma vez que, como já dissemos, compartilha o seu *kernel*, o que proporciona melhor desempenho por conta do gerenciamento único dos recursos.

Na maioria dos casos, a imagem de um *container* é bastante enxuta, havendo somente o necessário para o funcionamento da aplicação, que, quando em execução, possui um pequeno *overhead* se comparada à mesma aplicação rodando nativamente no sistema operacional, grande parte disso por conta do compartilhamento dos recursos com a máquina *host*.

Quando estamos utilizando máquinas virtuais, nós emulamos um novo sistema operacional e todo o seu hardware utilizando mais recursos da máquina *host*, o que não ocorre quando utilizamos *containers*, pois os recursos são compartilhados. O ganho óbvio disso é a capacidade de rodar mais *containers* em um único *host*, se comparado com a quantidade que se conseguiria com máquinas virtuais.

A seguir, na figura, podemos notar as diferenças de quando temos aplicações sendo executadas nativamente, máquinas virtuais e por fim em *containers*. Repare que não é necessário emular um novo sistema operacional quando estamos utilizando *container*, diferentemente das máquinas virtuais.



Outro ponto interessante na utilização de *containers* é a portabilidade. Não importa em qual ambiente você criou o seu *container*, ele irá rodar em qualquer outro que possua, no nosso caso, o Docker instalado, seja ele no Linux, MacOS ou Windows. Você não precisa se preocupar com suas dependências, está tudo dentro do *container*. :D

O desenvolvedor consegue, em sua própria máquina, criar a sua aplicação em *container* e depois executá-la em um servidor em produção sem nenhum problema de dependência ou algo do tipo – nem mesmo o bom e velho “engraçado, na minha máquina funciona” escapa, hein!

Lembre-se: o propósito da máquina virtual é emular um novo sistema operacional dentro do sistema operacional do *host*. Já o propósito do *container* é emular somente as aplicações e suas dependências e torná-las portáteis.

1.2. E quando começou que eu não vi?

Apesar de o termo ter se tornado *hype* nos últimos anos, durante décadas já utilizávamos *containers* em sistemas Unix através do comando *chroot* – sim, bem mais simplório, é verdade, pois era apenas uma forma de isolar o *filesystem*, mas já era o começo!

Em seguida vieram os *jails* do FreeBSD, que, além do isolamento do *filesystem*, permitiam também o isolamento de processos, seguidos de perto pela Sun, que desenvolveu o *Solaris Zones*, mais uma solução baseada em *containers*, porém somente para sistemas Solaris.

O grande passo rumo ao cenário que temos hoje foi a criação, pela Parallels do Virtuozzo, de um painel que permitia o fácil gerenciamento de *containers* e a disponibilização do *core* do Virtuozzo como *open source* com o nome de OpenVZ.

O OpenVZ foi uma ferramenta que ganhou bastante destaque no gerenciamento de *containers* e ajudou e muito na popularização do VPS (*Virtual Private Server*) e, conseqüentemente, na criação de centenas de empresas de *hosting* espalhadas pelo mundo. O principal ponto negativo do OpenVZ era a necessidade de aplicar um *patch* no *kernel* Linux.

Logo após surgir o OpenVZ, o Google iniciou o desenvolvimento do CGroups para o *kernel* do Linux e iniciou a utilização de *containers* em seus *datacenters*.

Em 2008 iniciou o projeto LXC, que trazia consigo o CGroups, *namespaces* e *chroot* para prover uma completa e estável solução para a criação e o gerenciamento de *containers*.

Porém, foi no ano de 2013 que os *containers* conquistaram o *mainstream*, saíram do *underground* através da utilização massiva pelas empresas de internet e gigantes de tecnologia e invadiram os principais eventos de tecnologia ao redor do mundo, com palestras sobre o sucesso na utilização de *containers* e com o melhor aproveitamento dos recursos físicos como CPU e memória, maior agilidade no *deployment* de novas aplicações, *start/stop* de *containers* em fração de segundos e tudo isso com uma facilidade que impressiona. Amigo, estamos falando do simplesmente sensacional **Docker**.

2. O que é o Docker?

2.1. Onde entra o Docker nessa história?

Tudo começou em 2008, quando Solomon Hykes fundou a dotCloud, empresa especializada em PaaS com um grande diferencial: o seu *Platform-as-a-Service* não era atrelado a nenhuma linguagem de programação específica, como era o caso, por exemplo, da Heroku, que suportava somente aplicações desenvolvidas em Ruby.

A grande virada na história da dotCloud ocorreu em março de 2013, quando decidiram tornar *open source* o *core* de sua plataforma – assim nascia o Docker!

As primeiras versões do Docker nada mais eram do que um *wrapper* do LXC integrado ao *Union Filesystem*, mas o seu crescimento foi fantástico e muito rápido, tanto que em seis meses seu *GitHub* já possuía mais de seis mil *stars* e mais de 170 pessoas contribuindo para o projeto ao redor do mundo.

Com isso, a dotCloud passou a se chamar Docker e a versão 1.0 foi lançada apenas 15 meses após sua versão 0.1. A versão 1.0 do Docker trouxe muito mais estabilidade e foi considerada “production ready”, além de trazer o Docker Hub, um repositório público para *containers*.

Por ser um projeto *open source*, qualquer pessoa pode visualizar o código e contribuir com melhorias para o Docker. Isso traz maior transparência e faz com que correções de *bugs* e melhorias aconteçam bem mais rápido do que seria em um software proprietário com uma equipe bem menor e poucos cenários de testes.

Quando o Docker 1.0 foi lançado e anunciado que estava pronto para produção, empresas como Spotify já o utilizavam em grande escala; logo AWS e Google começaram a oferecer suporte a Docker em suas nuvens. Outra gigante a se movimentar foi a Red Hat, que se tornou uma das principais parceiras do Docker, inclusive o incorporando-o ao *OpenShift*.

Atualmente, o Docker é oficialmente suportado apenas em máquinas Linux 64 *bits*. Isso significa que seus *containers* também terão que ser um Linux 64 *bits*, pois lembre que o *container* utiliza o mesmo *kernel* da máquina host. ;)

No final de 2014, a Microsoft anunciou que futuras versões do Windows Server suportarão o Docker, uma mudança significativa no seu modelo de negócio.

Suporte a outras plataformas como Solaris e BSD deverão acontecer em breve.

2.2. E esse negócio de camadas?

2.2.1. *Copy-On-Write* (COW) e Docker

Antes de entender as camadas propriamente ditas, precisamos entender como um dos principais requisitos para essa coisa acontecer, o *Copy-On-Write* (ou COW para os íntimos), funciona. Nas palavras do próprio Jérôme Petazzoni:

It's a little bit like having a book. You can make notes in that book if you want, but each time you approach the pen to the page, suddenly someone shows up and takes the page and makes a xerox copy and hand it back to you, that's exactly how copy on write works.

Em tradução livre, seria como se você tivesse um livro e que fosse permitido fazer anotações nele caso quisesse, porém, cada vez que você estivesse prestes a tocar a página com a caneta, de repente alguém aparecesse, tirasse uma xerox dessa página e entregasse a cópia para você. É exatamente assim que o *Copy-On-Write* funciona.

Basicamente, significa que um novo recurso, seja ele um bloco no disco ou uma área em memória, só é alocado quando for modificado.

Tá, mas o que isso tudo tem a ver com o Docker? Bom, como você sabe, o Docker usa um esquema de camadas, ou *layers*, e para montar essas camadas são usadas técnicas de *Copy-On-Write*. Um *container* é basicamente uma pilha de camadas compostas por N camadas *read-only* e uma, a superior, *read-write*.

2.3. Storage drivers

Storage drivers são *filesystems* que o *daemon* do Docker vai usar para gerenciar os *containers* e as imagens. O Docker aceita nativamente alguns *storage drivers*. A seguir, vamos falar brevemente sobre os principais.

2.3.1. AUFS (Another Union File System)

O primeiro *filesystem* disponível para o Docker foi o AUFS, um dos mais antigos *Copy-On-Write filesystems*, e inicialmente teve que passar por algumas modificações a fim de melhorar a estabilidade.

A ideia do AUFS é ter múltiplos diretórios (camadas) juntos em uma pilha única.

Quando você tenta ler um arquivo, a busca é iniciada pela camada superior, até achar o arquivo ou concluir que ele não existe. Para escrever em um arquivo, este precisa primeiro ser copiado para a camada superior (*writable*) – e, sim, você adivinhou: escrever em arquivos grandes pode causar certa degradação da performance.

Já que estamos falando de coisa chata, outra coisa que pode degradar a sua performance usando AUFS é o fato de que ele procura cada diretório de um *path* em cada camada do *filesystem* toda vez que você tentar executar um comando. Por exemplo, se você tem um *path* com cinco diretórios, serão realizadas 25 buscas (*stat()* *system call*). Isso pode ser bem complicado em aplicações que fazem *load* dinâmico, como os *apps* Python que importam os *.py* da vida.

Uma outra particularidade é quando algum arquivo é deletado. Quando isso acontece é criado um *whiteout* para esse arquivo. Em outras palavras, ele é renomeado para “.wh.arquivo”.

2.3.2. *Device Mapper*

Device Mapper é um *kernel-based framework* da Red Hat usado para algumas abstrações, como, por exemplo, o mapeamento de “blocos físicos” em “blocos lógicos”, permitindo técnicas como LVM e RAID. No contexto do Docker, porém, ele se resume ao “thin provisioning target”, ou ao *storage driver* “devicemapper”. Assim que essa coisa de Docker começou a andar, o pessoal da Red Hat (e toda a galera que usava alguma *distro* relacionada com Red Hat) teve bastante interesse, só que havia um problema: eles não queriam usar AUFS. Para resolver isso, eles reuniram uma equipe de engenheiros muito habilidosos que adicionaram suporte ao *Device Mapper* no Docker.

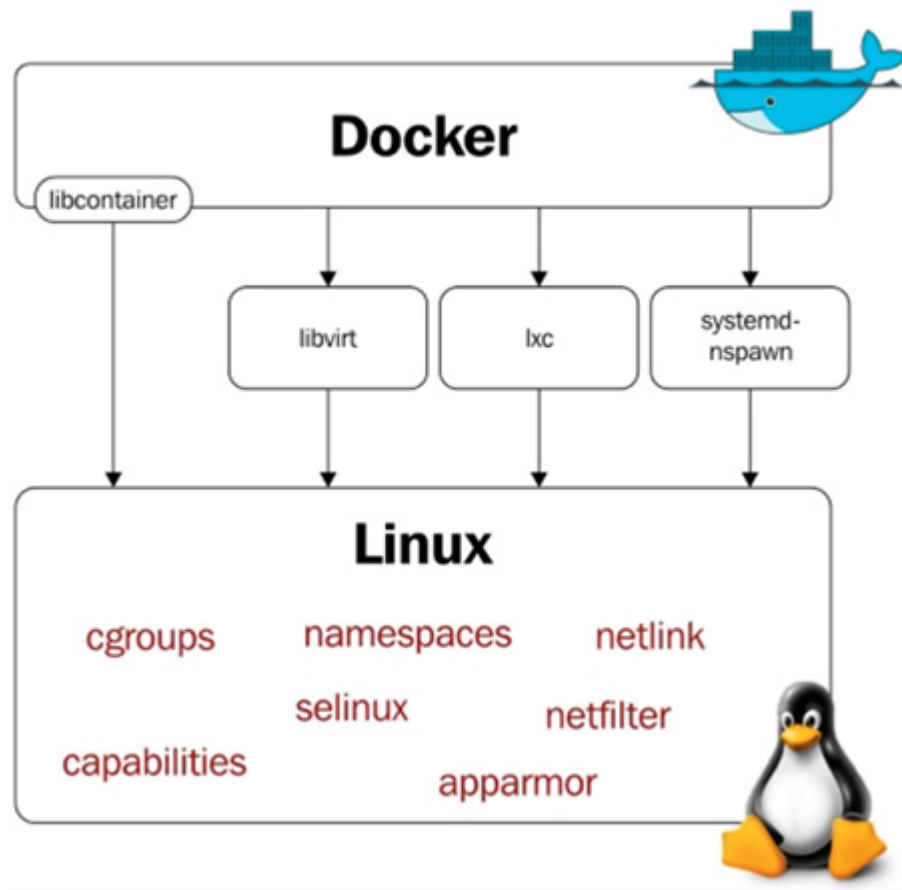
Em se tratando de Docker, o *Device Mapper* e o AUFS são bem similares: a grande diferença entre eles é que, no *Device Mapper*, quando você precisa escrever no arquivo, a cópia é feita em nível de blocos, não de *filesystem*, e com isso você ganha uma granularidade bem maior. Em teoria, o problema que você tinha quando escrevia um arquivo grande desaparece. Dito isso, você pode pensar que a performance no *Device Mapper* é melhor do que a do AUFS, só que existe um problema bem chato nele: por padrão, ele escreve em *loopback* (*sparse files*), deixando as operações relativamente mais lentas.

Além de AUFS e *Device Mapper*, você também pode usar BRTFS e OverlayFS como *storage driver*. Por serem tecnologias relativamente jovens, aprecie com moderação.

2.4. Docker Internals

O Docker utiliza algumas *features* básicas do *kernel* Linux para seu funcionamento. A seguir temos um diagrama onde é possível visualizar os

módulos e *features* do *kernel* de que o Docker faz uso:



2.5. Namespaces

Namespaces foram adicionados no *kernel* Linux na versão 2.6.24 e são eles que permitem o isolamento de processos quando estamos utilizando o Docker. São os responsáveis por fazer com que cada *container* possua seu próprio *environment*, ou seja, cada *container* terá a sua árvore de processos, pontos de montagens, etc., fazendo com que um *container* não interfira na execução de outro. Vamos saber um pouco mais sobre alguns *namespaces* utilizados pelo Docker.

2.5.1. PID namespace

O *PID namespace* permite que cada *container* tenha seus próprios identificadores de processos. Isso faz com que o *container* possua um PID para um processo em execução – e quando você procurar por esse processo na máquina *host* o encontrará; porém, com outra identificação, ou seja, com outro PID.

A seguir temos o processo “testando.sh” sendo executado no *container*. Perceba o PID desse processo na árvore de processos dele:

```
root@c774fa1d6083:/# bash testando.sh &
[1] 7
root@c774fa1d6083:/# ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 18:06 ? 00:00:00 /bin/bash
root 7 1 0 18:07 ? 00:00:00 bash testando.sh
root 8 7 0 18:07 ? 00:00:00 sleep 60
root 9 1 0 18:07 ? 00:00:00 ps -ef
root@c774fa1d6083:/#
```

Agora, perceba o PID do mesmo processo exibido agora através do *host*:

```
root@linuxtips:~# ps -ef | grep testando.sh
root 2958 2593 0 18:12 pts/2 00:00:00 bash testando.sh
root 2969 2533 0 18:12 pts/0 00:00:00 grep --color=auto
testando.sh
root@linuxtips:~#
```

Diferentes né? Porém, são o mesmo processo. :)

2.5.2. Net namespace

O *net namespace* permite que cada *container* possua sua interface de rede e portas. Para que seja possível a comunicação entre os *containers*, é necessário criar dois *net namespaces* diferentes, um responsável pela interface do *container* (normalmente utilizamos o mesmo nome das interfaces convencionais do

Linux, por exemplo, a `eth0`) e outro responsável por uma interface do *host*, normalmente chamada de `veth*` (`veth` + um identificador aleatório). Essas duas interfaces estão *linkadas* através da *bridge* `Docker0` no *host*, que permite a comunicação entre os *containers* através de roteamento de pacotes.

Conforme falamos, veja as interfaces.

Interfaces do *host*:

```
root@linuxtips:~# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP group default qlen 1000
    link/ether 00:1c:42:c7:bd:d8 brd ff:ff:ff:ff:ff:ff
    inet 10.211.55.35/24 brd 10.211.55.255 scope global eth1
        valid_lft forever preferred_lft forever
    inet6 fdb2:2c26:f4e4:0:21c:42ff:fec7:bdd8/64 scope global dynamic
        valid_lft 2591419sec preferred_lft 604219sec
    inet6 fe80::21c:42ff:fec7:bdd8/64 scope link
        valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:c7:c1:37:14 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:c7ff:fe1:3714/64 scope link
        valid_lft forever preferred_lft forever
5: vetha2e1681: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue master docker0 state UP group default
    link/ether 52:99:bc:ab:62:5e brd ff:ff:ff:ff:ff:ff
    inet6 fe80::5099:bcff:feab:625e/64 scope link
        valid_lft forever preferred_lft forever
root@linuxtips:~#
```

Interfaces do *container*:

```

root@6ec75484a5df:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
6: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link
        valid_lft forever preferred_lft forever
root@6ec75484a5df:/#

```

Conseguiu visualizar as interfaces Docker0 e veth* do *host*? E a eth0 do *container*? Sim? *Otémooo!* :D

2.5.3. *IPC namespace*

É evolução do *chroot*. Com o *mnt namespace* cada *container* pode ser dono de seu ponto de montagem, bem como de seu sistema de arquivos raiz. Ele garante que um processo rodando em um sistema de arquivos montado pelo *mnt namespace* não consiga acessar outro sistema de arquivos montado por outro *mnt namespace*.

2.5.4. *Mnt namespace*

Ele provê um SystemV IPC isolado, além de uma fila de mensagens POSIX própria.

2.5.5. *UTS namespace*

Responsável por prover o isolamento de *hostname*, nome de domínio, versão do SO, etc.

2.5.6. User namespace

O mais recente *namespace* adicionado no *kernel* Linux, disponível desde a versão 3.8. É o responsável por manter o mapa de identificação de usuários em cada *container*.

2.6. Cgroups

É o *cgroups* o responsável por permitir a limitação da utilização de recursos do *host* pelos *containers*.

Com o *cgroups* você consegue gerenciar a utilização de CPU, memória, dispositivos, I/O, etc.

2.7. Netfilter

A ferramenta *iptables* faz parte de um módulo chamado *netfilter*. Para que os *containers* consigam se comunicar, o Docker constrói diversas regras de roteamento através do *iptables*; inclusive utiliza o NAT, que veremos mais adiante no livro.

```
root@linuxtips:~# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source      destination
DOCKER     all  --  anywhere    anywhere
ADDRTYPE match dst-type LOCAL
```

```
Chain INPUT (policy ACCEPT)
target     prot opt source      destination
```

```
Chain OUTPUT (policy ACCEPT)
target     prot opt source      destination
DOCKER     all  --  anywhere    !127.0.0.0/8
ADDRTYPE match dst-type LOCAL
```

Chain POSTROUTING (policy ACCEPT)

```
target      prot opt source      destination
MASQUERADE  all  --  172.17.0.0/16  anywhere
```

Chain DOCKER (2 references)

```
target prot opt source destination
RETURN all  --  anywhere anywhere
```

```
root@linuxtips:~#
```

2.8. Para quem ele é bom?

O Docker é muito bom para os desenvolvedores, pois com ele você tem liberdade para escolher a sua linguagem de programação, seu banco de dados e sua distribuição predileta. Já para os *sysadmins* é melhor ainda, pois, além da liberdade de escolher a distribuição, não precisamos preparar o servidor com todas as dependências da aplicação. Também não precisamos nos preocupar se a máquina é física ou virtual, pois o Docker suporta ambos.

A empresa como um todo ganha com a utilização do Docker, pois dá maior agilidade no processo de desenvolvimento de aplicações, encurtando o processo de transição entre os ambientes de DEV, QA e PROD, pois é utilizado o mesmo *container*. Traz menos custos com hardware por conta do melhor gerenciamento e aproveitamento dos recursos, além do *overhead*, que é bem menor se comparado com outras soluções, como a virtualização.

Com Docker fica muito mais viável a criação de *microservices* (microserviços, a ideia de uma grande aplicação ser quebrada em várias pequenas partes e estas executarem tarefas específicas), um assunto que tem ganhado cada vez mais espaço no mundo da tecnologia e que vamos abordar com mais detalhes no final deste livro.

Ainda temos diversos outros motivos para utilizar *containers* e que vamos descobrindo conforme evoluímos com a utilização do Docker. :D

3. Instalando o Docker

3.1. Quero instalar, vamos lá?

Bom, dado que você já sabe o que é um *container* e o que é o tal do Docker, chegou a hora de pôr a mão na massa. Vamos instalar o Docker pela primeira vez!

O *daemon* do Docker roda nativo em distribuições Linux, e por isso a instalação em sistemas operacionais que não sejam Linux consiste basicamente em subir uma VM e rodar o *daemon* de lá. O cliente, no entanto, pode ser instalado nos principais sistemas operacionais disponíveis atualmente.

Para realizar a instalação do Docker em máquinas Linux é bastante simples. Precisamos somente observar alguns pontos:

- ⇒ O Docker não suporta processadores 32 *bits*.
- ⇒ O Docker é suportado somente (*stable*) na versão do *kernel* 3.8 ou superior.
- ⇒ O *kernel* deve ter suporte aos sistemas de arquivos utilizados pelo Docker, como o AUFS, *Device Mapper*, OverlayFS, etc.
- ⇒ O *kernel* deverá ter suporte a *cgroups* e *namespaces*, o que normalmente já vem por *default* habilitado na maioria das *distros*.

Você também pode acessar a URL:
<https://docs.docker.com/engine/security/security/>.

Lá é possível aprender a instalar o Docker em diversas distribuições Linux, nos principais *clouds* e também no MacOS e no Windows.

Neste livro vamos utilizar a distribuição Debian Linux 8, porém o processo de instalação não muda quase nada para outras distribuições. Chega de conversa, vamos lá!

Primeiro, vamos verificar a versão do *kernel* para saber se ele é compatível com o Docker:

```
# uname -r
```

Agora, a instalação do Docker é bastante simples. Você pode optar por instalá-lo utilizando os pacotes disponíveis para sua *distro* – por exemplo, o *apt-get* ou *yum*.

Nós preferimos fazer a instalação através da execução do *curl* a seguir, que irá baixar um *script* de instalação e já o executará:

```
# curl -fsSL https://get.docker.com/ | sh
```

Assim ele sempre buscará a versão mais recente do Docker. :)

Agora adicione a chave *pgp* para conseguir baixar o pacote do repositório do Docker.

Caso você esteja utilizando o Debian e queira realizar a instalação através dos pacotes disponíveis no repositório, faça:

```
#apt-key adv --keyserver \ hkp://pgp.mit.edu:80 --recv-keys \
58118E89F3A912897C070ADB76221572C52609D
```

Agora vamos criar/editar o arquivo “/etc/apt/sources.list.d/docker.list” e adicionar o endereço do repositório correspondente à versão do seu Debian. No nosso caso estamos utilizando a versão Debian 8, também conhecida como Jessie.

```
# vim /etc/apt/sources.list.d/docker.list
# Debian Jessie
deb https://apt.dockerproject.org/repo debian-jessie main
```

Após adicionar a linha anterior, é necessário atualizar a lista de repositórios executando:

```
# apt-get update
```

Após finalizar a atualização da lista de repositórios disponíveis, já podemos fazer a instalação do Docker. O nome do pacote é “docker-engine”. :)

```
# apt-get install docker-engine
```

Vamos verificar se o Docker está em execução. Digite na linha de comando o seguinte:

```
#!/etc/init.d/docker status
```

Ou:

```
# service docker status
```

```
docker stop/waiting
```

Com isso, podemos verificar se o processo está em execução. Como podemos notar, o *daemon* do Docker não está em execução, portanto vamos iniciá-lo.

```
# service docker start
```

```
docker start/running, process 4303
```

```
# service docker status
```

```
docker start/running, process 4303
```

Perfeito! Agora já temos o Docker instalado e pronto para começar a brincar com os *containers*. \o/

3.1.1. Dica importante

Por padrão, o *daemon* do Docker faz *bind* em um *socket* Unix, e não em uma porta TCP. *Sockets* Unix, por sua vez, são de propriedade e de uso exclusivo do usuário *root* (por isso o Docker sempre é iniciado como *root*), mas também podem ser acessados através do *sudo* por outros usuários.

Para evitar que você tenha que ficar usando *sudo* ao rodar comandos do Docker, crie um grupo chamado *docker* e adicione o seu usuário a ele. Pare o serviço e inicie-o novamente.

Infelizmente, nem tudo são flores. Esse procedimento faz com que o usuário tenha **os mesmos privilégios do usuário *root*** em operações relacionadas ao Docker. Mais informações no link: <https://docs.docker.com/engine/security/security/>.

Para criar um grupo no Linux e adicionar um usuário não tem segredo, basta rodar:

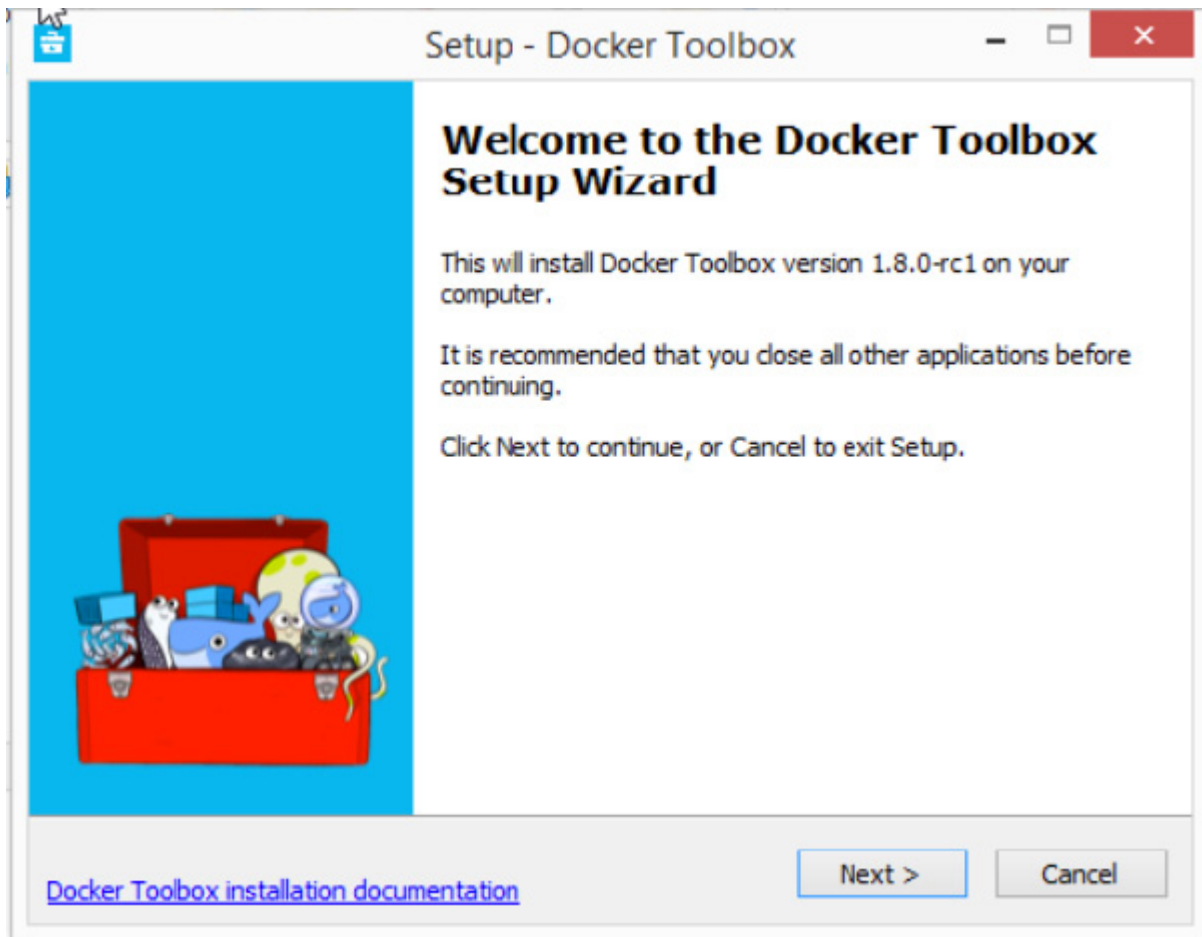
```
$ sudo usermod -aG docker user
```

Dica de um milhão de dólares: **user = seu usuário**. :D

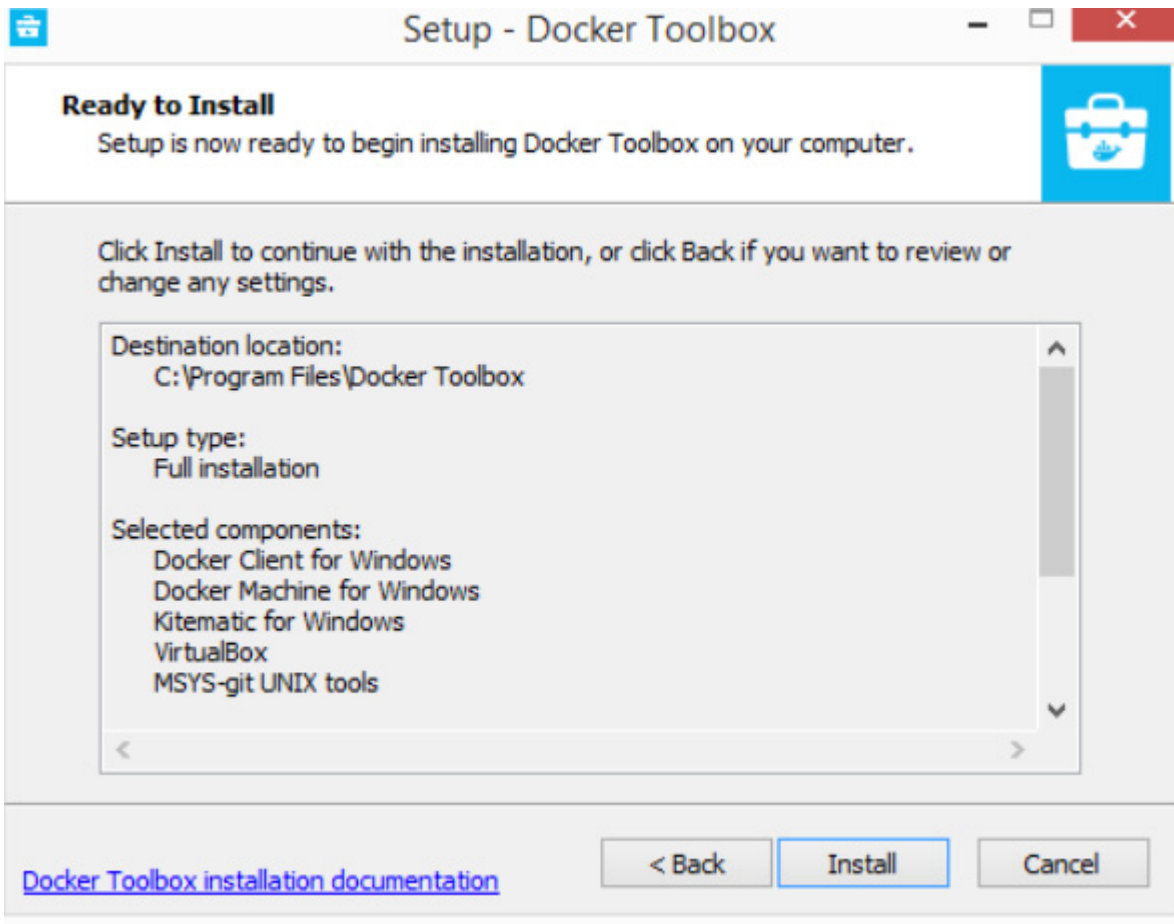
3.2. Windows, MacOS, etc.

Bem, se você não usa Linux hoje (considere essa possibilidade!), a única forma de usar o Docker no seu ambiente de testes/desenvolvimento é instalar uma máquina virtual com Linux e nela instalar um Docker. Para facilitar esse processo foi criado o **Docker Toolbox**, que pode ser baixado em <https://www.docker.com/toolbox>. Uma vez instalado, você terá disponível:

- ⇒ Oracle VirtualBox.
- ⇒ Docker Engine.
- ⇒ Docker Machine.
- ⇒ Kitematic, um GUI para o Docker.
- ⇒ Um *shell* personalizado para trabalhar em um ambiente Docker.



A instalação é bem simples, basicamente o velho *next, install, finish*.





Lembrando que no Windows ele roda da versão 7 em diante e no MacOS a partir da versão 10.8, a *Mountain Lion*.

4. Executando e administrando *containers* Docker

4.1. Então vamos brincar com esse tal de *container*!

Como todos sabemos, o Docker utiliza a linha de comando para que você possa interagir com ele – basicamente você utiliza o comando “docker”.

Bom, agora que já iniciamos o Docker, vamos rodar nosso primeiro *container*.

Como é de costume quando alguém está aprendendo uma nova linguagem de programação, é bem comum fazer como o primeiro código um *hello world*!

Apesar de o Docker não ser uma linguagem de programação, vamos utilizar esse costume com o nosso primeiro exemplo de um *container* em execução.

O Docker possui uma imagem personalizada de *hello-world* e serve para que você possa testar a sua instalação e validar se tudo funciona conforme o esperado. :D

Para que possamos executar um *container*, utilizamos o parâmetro “run” do comando “docker”:

```
root@linuxtips:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

03f4658f8b78: Pull complete
a3ed95caeb02: Pull complete
Digest:
```

```
sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
```

```
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker.
```

```
This message shows that your installation appears to be working correctly.
```

```
To generate this message, Docker took the following steps:
```

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

```
To try something more ambitious, you can run an Ubuntu container with:
```

```
$ docker run -it ubuntu bash
```

```
Share images, automate workflows, and more with a free Docker Hub account:
```

```
https://hub.docker.com
```

```
For more examples and ideas, visit:
```

```
https://docs.docker.com/userguide/
```

```
root@linuxtips:~#
```

No exemplo anterior, estamos executando um *container* utilizando a imagem personalizada do *hello-world*.

Apesar de ser uma tarefa simples, quando você executou o comando “docker run hello-world” foram necessárias quatro etapas para sua conclusão, vamos ver quais:

1. O comando “docker” se comunica com o *daemon* do Docker informando a ação desejada.
2. O *daemon* do Docker verifica se a imagem “hello-world” encontra-se em seu *host*; caso ainda não, o Docker faz o *download* da imagem diretamente

do Docker Hub.

3. O *daemon* do Docker cria um novo *container* utilizando a imagem que você acabou de baixar.
4. O *daemon* do Docker envia a saída para o comando “docker”, que imprime a mensagem em seu terminal.

Viu? É simples como voar! :)

Muito bem, agora que nós já temos uma imagem em nosso *host*, como eu faço para visualizá-la?

Muito simples, basta digitar o seguinte comando:

```
root@linuxtips:~# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
hello-world latest 690ed74de00f 5 months 960 B
root@linuxtips:~#
```

Como você pode notar no código, a saída traz cinco colunas:

- ⇒ **REPOSITORY** – O nome da imagem.
- ⇒ **TAG** – A versão da imagem.
- ⇒ **IMAGE ID** – Identificação da imagem.
- ⇒ **CREATED** – Quando ela foi criada.
- ⇒ **SIZE** – Tamanho da imagem.

Quando executamos o comando “docker run hello-world”, ele criou o *container*, imprimiu a mensagem na tela e depois o *container* foi finalizado automaticamente, ou seja, ele executou sua tarefa, que era exibir a mensagem, e depois foi finalizado.

Para ter certeza de que ele realmente foi finalizado, digite:

```
root@linuxtips:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORT NAMES
root@linuxtips:~#
```

Com o “docker ps”, você consegue visualizar todos os *containers* em execução e ainda obter os detalhes sobre eles. A saída do “docker ps” é dividida em sete colunas; vamos conhecer o que elas nos dizem:

- ⇒ **CONTAINER ID** – Identificação única do *container*.
- ⇒ **IMAGE** – A imagem que foi utilizada para a execução do *container*.
- ⇒ **COMMAND** – O comando em execução.
- ⇒ **CREATED** – Quando ele foi criado.
- ⇒ **STATUS** – O seu status atual.
- ⇒ **PORTS** – A porta do *container* e do *host* que esse *container* utiliza.
- ⇒ **NAMES** – O nome do *container*.

Uma opção interessante do “docker ps” é o parâmetro “-a”.

```
root@linuxtips:~# docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS
PORTS          NAMES
6e45cf509282   hello-world "/hello"               4seconds      Exited(0)
traced_ardinghell
root@linuxtips:~#
```

Com a opção “-a” você consegue visualizar não somente os *containers* em execução, como também *containers* que estão parados ou que foram finalizados.

4.2. Legal, quero mais!

Agora que vimos como criar um simples *container*, bem como visualizar as imagens e *containers* que estão em nosso *host*, vamos criar um novo, porém conhecendo três parâmetros que irão trazer maior flexibilidade no uso e na administração de nossos *containers*. Estou falando dos parâmetros “-t”, “-i” e “-d”.

- ⇒ **-t** – Disponibiliza um TTY (console) para o nosso *container*.

- ⇒ **-i** – Mantém o STDIN aberto mesmo que você não esteja conectado no *container*.
- ⇒ **-d** – Faz com que o *container* rode como um *daemon*, ou seja, sem a interatividade que os outros dois parâmetros nos fornecem.

Com isso temos dois modos de execução de nossos *containers*: modo interativo ou *daemonizando* o *container*.

4.2.1. Modo interativo

Na maior parte das vezes você vai subir um *container* a partir de uma imagem que já está pronta, toda ajustadinha. Porém, há alguns casos em que você precisa interagir com o seu *container* – isso pode acontecer, por exemplo, na hora de montar a sua imagem personalizada.

Nesse caso, usar o modo interativo é a melhor opção. Para isso, basta passar os parâmetros “-ti” ao comando “docker run”.

```
root@linuxtips:~# docker run -ti ubuntu /bin/bash
[root@4116917387a6 /]#
```

4.2.2. Daemonizando o container

Utilizando o parâmetro “-d” do comando “docker run”, é possível *daemonizar* o *container*, fazendo com que o *container* seja executado como um processo *daemon*.

Isso é ideal quando nós já possuímos um *container* que não iremos acessar (via *shell*) para realizar ajustes. Imagine uma imagem já com a sua aplicação e tudo que precisa configurado; você irá subir o *container* e somente irá consumir o serviço entregue por sua aplicação. Se for uma aplicação *web*, basta acessar no *browser* passando o IP e a porta onde o serviço é disponibilizado no *container*. Sensacional, não?

```
root@linuxtips:~# docker run -d minha_imagem_pronta
49f9014740042fc9d0543d7edb6ecbd711def79d6fef7bb5de5988ad7082d6cf
```

Ou seja, se você quer subir um *container* para ser utilizado como uma máquina Linux convencional com *shell* e que necessita de alguma configuração ou ajuste, utilize o modo interativo, ou seja, os parâmetros “-ti”.

Agora, se você já tem o *container* configurado, com sua aplicação e todas as dependências sanadas, não tem a necessidade de usar o modo interativo – nesse caso utilizamos o parâmetro “-d”, ou seja, o *container* *daemonizado*. Vamos acessar somente os serviços que ele provê, simples assim. :D

4.3. Entendi, agora vamos praticar um pouco?

Perfeito. Vamos iniciar um novo *container* utilizando dois desses novos parâmetros que aprendemos.

Para o nosso exemplo, vamos subir um *container* do Centos 7:

```
root@linuxtips:~# docker run -ti centos:7
Unable to find image 'centos:7' locally
7: Pulling from library/centos

a3ed95caeb02: Pull complete
196355c4b639: Pull complete
Digest:
sha256:3cdc0670fe9130ab3741b126cfac6d7720492dd2c1c8ae033dcd77d32855bab
2
Status: Downloaded newer image for centos:7
[root@3c975fb7fbb5 /]#
```

Como a imagem não existia em nosso *host*, ele fez o *download* do Docker Hub. Caso ela já estivesse lá, seria utilizada.

Perceba que mudou o seu *prompt* (variável \$PS1), pois agora você já está dentro do *container*. Para provar que estamos dentro do nosso *container* Centos, execute o seguinte comando:

```
[root@3c975fb7fbb5 /]# cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core)
[root@3c975fb7fbb5 /]#
```

O arquivo “/etc/redhat-release” indica qual a versão do Centos que estamos utilizando, ou seja, estamos realmente em nosso *container* Centos 7. :D

4.4. Tá, agora quero sair...

Um detalhe importante é que você não pode utilizar o comando “exit” para sair do console, pois dessa forma você automaticamente dá *stop* no *container*. Caso queira sair de um *container* e mantê-lo em execução é necessário sair com o seguinte atalho do teclado:

mantenha o botão Ctrl pressionado + p + q

Assim, você sairá do *container* e ele continuará em execução. Para confirmar se o *container* continua em execução, faça:

```
root@linuxtips:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS
PORTS NAMES
3c975fb7fbb5 centos:7 "/bin/bash" 2minutes Up 2 minutes
angry_wescoff
root@linuxtips:~#
```

4.5. Posso voltar ao *container*?

Deixamos o nosso *container* em execução e agora queremos acessá-lo novamente. Como podemos fazer?

Simples! Basta digitar o seguinte comando:

```
root@linuxtips:~# docker attach <CONTAINER ID>
```

O parâmetro “attach” do comando “docker” possibilita nos conectarmos a um *container* em execução. Para isso, basta passar como parâmetro o “CONTAINER ID”, que você consegue através da saída do “docker ps”, conforme mostramos no exemplo anterior.

4.6. Continuando com a brincadeira...

Existe a possibilidade de criar um *container*, porém não executá-lo imediatamente. Quando fazemos o uso do parâmetro “create” do comando “docker”, ele apenas cria o *container*, não o inicializando, conforme notamos no exemplo a seguir:

```
root@linuxtips:~# docker create ubuntu
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu

5a132a7e7af1: Pull complete
fd2731e4c50c: Pull complete
28a2f68d1120: Pull complete
a3ed95caeb02: Pull complete
Digest:
sha256:4e85ebe01d056b43955250bbac22bdb8734271122e3c78d21e55ee235fc6802
d
Status: Downloaded newer image for ubuntu:latest
3e63e65db85a6e36950959dc6bdc00279e2208a335580c478e01723819de9467
root@linuxtips:~#
```

Perceba que quando você digita “docker ps” ele não traz o *container* recém-criado, afinal a saída do “docker ps” somente traz os *containers* em execução. Para visualizar o *container* recém-criado foi necessário utilizar o parâmetro “-a”.

```
root@linuxtips:~# docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS
PORTS         NAMES
3e63e65db85a   ubuntu   "/bin/bash"             18 seconds ago Created
elo_visves
root@linuxtips:~#
```

Para que o nosso *container* recém-criado seja executado, basta utilizar o “docker run”, conforme segue:

```
root@linuxtips:~# docker run -ti ubuntu
root@b422f04df14c:/#
```

Verificando se estamos realmente utilizando o *container* do Ubuntu:

```
root@b422f04df14c:/# cat /etc/debian_version
jessie/sid
root@b422f04df14c:/#
```

Lembrando que para sair do *container* e mantê-lo em execução é necessário utilizar o atalho: **Ctrl + p + q**.

4.7. Subindo e baixando *containers*...

Caso eu queira parar um *container* em execução, basta utilizar o parâmetro “stop” seguido do “CONTAINER ID”:

```
# docker stop [CONTAINER ID]
```

Verificando se o *container* continua em execução:

```
# docker ps
```

De novo... para visualizar os *containers* que não estão em execução é necessário utilizar o parâmetro “-a”.

Para colocar novamente em execução um *container* que está parado, é necessário utilizar o parâmetro “start” do comando “docker” seguido do “CONTAINER ID”:

```
# docker start [CONTAINER ID]
```

Da mesma forma como podemos utilizar o *stop/start* para desligar/iniciar um *container*, podemos também fazer o uso do “restart”, como notamos a seguir:

```
# docker restart [CONTAINER ID]
```

Para pausar um *container*, execute:

```
# docker pause [CONTAINER ID]
```

E verifique o status do *container*:

```
root@linuxtips:~# docker ps
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS
PORTS  NAMES
```

```
b34f4987bdce ubuntu "/bin/bash" 12 seconds ago Up
11seconds(Paused) drunk_turi
root@linuxtips:~#
```

Para “despausar” o *container*:

```
# docker unpause [CONTAINER ID]
```

4.8. Visualizando o consumo de recursos pelo *container*

Caso você queira visualizar informações referentes ao consumo de recursos pelo *container*, também é bastante simples: basta utilizar o parâmetro “stats” para verificar o consumo de CPU, memória e rede pelo *container*.

```
# docker stats [CONTAINER ID]
CONTAINER CPU% MEM USAGE / LIMIT MEM % NET I/O
BLOCK I/O
b34f4987bdce 0.00% 503.8 kB / 2.094 GB 0.02% 648 B /
648 B 0 B / 0 B
```

Agora, se você quer visualizar quais processos estão em execução em determinado *container*, utilize o parâmetro “top”. Com ele você consegue informações sobre os processos em execução, como, por exemplo, UID e o PID do processo.

```
# docker top [CONTAINER ID]
UID PID PPID C STIME TTY TIME CMD
root 10656 4303 0 20:24 pts/3 00:00:00 /bin/bash
```

Para verificar os *logs* de um determinado *container*, utilize o parâmetro “logs”, simples assim. :D

```
# docker logs [CONTAINER ID]
```

Lembre-se: ele exibe o STDOUT, a saída padrão. Ou seja, normalmente você irá visualizar o histórico de mensagens que aparecerem em primeiro plano

durante a execução do *container*.

4.9. Cansei de brincar de *container*, quero removê-lo!

Bem, remover um *container* é mais simples ainda do que sua criação. Quando removemos um *container*, a imagem que foi utilizada para a sua criação permanece no *host*; somente o *container* é apagado.

```
root@linuxtips:~# docker rm b34f4987bdce
Failed to remove container (b34f4987bdce): Error response from daemon:
Conflict, You cannot remove a running container. Stop the container
before attempting removal or use -f
root@linuxtips:~#
```

Perceba que, quando você tentou remover o *container*, ele retornou um erro dizendo que falhou em remover, pois o *container* estava em execução. Ele inclusive recomenda que você pare o *container* antes de removê-lo ou então utilize a opção “-f”, forçando assim sua remoção.

```
root@linuxtips:~# docker rm -f b34f4987bdce
b34f4987bdce
root@linuxtips:~#
```

Para confirmar a remoção do *container*, utilize o comando “docker ps”.

5. Configurando CPU e memória para os meus *containers*

Vamos imaginar que você precise subir quatro *containers* para um projeto novo. Esses *containers* possuem as seguintes características:

⇒ Dois *web servers*.

⇒ Dois DB MySQL.

Evidentemente, por se tratar de serviços diferentes, na maioria dos casos possuem características de consumo de recursos, como CPU e memória, diferentes um do outro.

⇒ *Web server* – Dois CPUs | 512 MB de memória

⇒ DB MySQL – Quatro CPUs | 2 GB de memória

E agora, como fazemos? :(

Por padrão, quando você executa um *container* sem especificar a quantidade de recursos que ele irá utilizar, ele sobe sem um controle, podendo inclusive impactar o *host* onde está sendo executado.

Portanto, é muito importante limitar a utilização de recursos de seus *containers*, visando um melhor aproveitamento de seus recursos computacionais, como veremos agora. :)

5.1. Especificando a quantidade de memória

Primeiro, vamos executar um *container* para realizarmos o nosso exemplo.

```
root@linuxtips:~# docker run -ti --name teste debian
```

Agora vamos visualizar a quantidade de memória que está configurada para esse *container*. Uma forma fácil é utilizar a saída do comando “docker inspect”:

```
root@linuxtips:~# docker inspect teste | grep -i mem
  "CpusetMems": "",
  "KernelMemory": 0,
  "Memory": 0,
  "MemoryReservation": 0,
  "MemorySwap": 0,
  "MemorySwappiness": -1,
```

Como percebemos, os valores correspondentes à memória estão zerados, ou seja, sem nenhum limite estabelecido.

Vamos agora subir um novo *container*, porém passando os valores para que utilize 512 MB de memória:

```
root@linuxtips:~# docker run -ti -m 512M --name
novo_container debian
```

Utilizamos o parâmetro “-m” para especificar a quantidade de memória que desejamos disponibilizar ao *container*. Vamos utilizar o “docker inspect” novamente para verificar se a nossa configuração funcionou:

```
root@linuxtips:~# docker inspect novo_container | grep -i mem
  "CpusetMems": "",
  "KernelMemory": 0,
  "Memory": 536870912,
  "MemoryReservation": 0,
  "MemorySwap": -1,
  "MemorySwappiness": -1,
```

Muito bom, parece que deu certo!

Observe o campo “Memory”, onde temos o valor que passamos no momento em que criamos o *container*. Vale ressaltar que o valor exibido é em *bytes*.

Quando você utiliza o parâmetro “-m” ou “--memory”, você está passando o máximo de memória que o *container* utilizará do *host*.

5.2. Especificando a quantidade de CPU

Para que possamos configurar a utilização de CPU do *host* pelo *container*, precisamos antes entender como funciona a divisão do consumo entre os *containers*.

Por padrão, todos os *containers* podem consumir os recursos de CPU de forma idêntica, na mesma proporção. Porém, quando você utiliza o parâmetro “--cpu-shares” você consegue redefinir essa proporção. O valor padrão do “--cpu-shares” é 1024, ou seja, todos os *containers*, quando não especificado, iniciam com esse valor.

Para que entenda melhor, imagine que você possui três *containers*, sendo que o *container* “A” possui um “cpu-share” de 1024, enquanto os *containers* “B” e “C” possuem um “cpu-share” de 512. Isso significa que o *container* “A” poderá utilizar 50% dos CPUs do *host* e os *containers* “B” e “C”, apenas 25% cada, entendeu? Fácil, né!?

Vamos criar esses *containers* utilizando o parâmetro “--cpu-shares”:

```
root@linuxtips:~# docker run -ti --cpu-shares 1024 --name teste1 debian
```

```
root@linuxtips:~# docker run -ti --cpu-shares 512 --name teste2 debian
```

```
root@linuxtips:~# docker run -ti --cpu-shares 512 --name teste3 debian
```

Para verificar, vamos utilizar o comando “docker inspect”:

```
root@linuxtips:~# docker inspect teste1 | grep -i cpu
  "CpuShares": 1024,
  "CpuPeriod": 0,
  "CpuQuota": 0,
  "CpusetCpus": "",
  "CpusetMems": "",
root@linuxtips:~# docker inspect teste2 | grep -i cpu
```

```
    "CpuShares": 512,  
    "CpuPeriod": 0,  
    "CpuQuota": 0,  
    "CpusetCpus": "",  
    "CpusetMems": "",  
root@linuxtips:~# docker inspect teste3 | grep -i cpu  
    "CpuShares": 512,  
    "CpuPeriod": 0,  
    "CpuQuota": 0,  
    "CpusetCpus": "",  
    "CpusetMems": "",
```

O campo “CpuShares” traz a informação que configuramos. :)

Simple, fácil e rápido!

5.3. Eu consigo alterar CPU e memória dos meus *containers* em execução?

Sim! \o/

Com o lançamento da versão 1.10 do Docker, temos o comando “docker update”, que permite alterar as configurações referentes a CPU, memória e IO com o *container* em execução de forma muito simples! Isso é fantástico!

Como exemplo, vamos subir um *container* e em seguida vamos alterar as informações referentes a memória e CPU:

```
root@linuxtips:~# docker run -ti --cpu-shares 1024 -m 512m --name  
teste1 debian
```

```
root@linuxtips:~# docker inspect teste1 | grep -i cpu  
    "CpuShares": 1024,  
    "CpuPeriod": 0,  
    "CpuQuota": 0,  
    "CpusetCpus": "",  
    "CpusetMems": "",  
root@linuxtips:~# docker inspect teste1 | grep -i mem  
    "CpusetMems": "",  
    "KernelMemory": 0,  
    "Memory": 536870912,
```



```
"MemoryReservation": 0,  
"MemorySwap": -1,  
"MemorySwappiness": -1,
```

Agora, vamos alterar os valores de limite de CPU e memória:

```
root@linuxtips:~# docker update -m 256m --cpu-shares 512 teste1  
teste1  
root@linuxtips:~# docker inspect teste1 | grep -i cpu  
    "CpuShares": 512,  
    "CpuPeriod": 0,  
    "CpuQuota": 0,  
    "CpusetCpus": "",  
    "CpusetMems": "",  
root@linuxtips:~# docker inspect teste1 | grep -i mem  
    "CpusetMems": "",  
    "KernelMemory": 0,  
    "Memory": 268435456,  
    "MemoryReservation": 0,  
    "MemorySwap": -1,  
    "MemorySwappiness": -1,
```

Funcionou? SIMMM!

Assim, com os *containers* em execução, mudamos as informações referentes a memória e CPU!

Existem outros parâmetros do “docker update”. Para verificar a lista completa, digite “docker update --help”.

6. Meu primeiro e tosco *dockerfile*...

Tudo que nós fizemos até agora foi escrever na linha de comando, o que é OK para aprender. Porém, principalmente nos dias de hoje, não dá para viver mais sem automatizar as coisas – se você, assim como nós, adora automatizar tudo que for possível, vai gostar bastante desse assunto.

O *dockerfile* nada mais é do que um arquivo onde você determina todos os detalhes do seu *container*, como, por exemplo, a imagem que você vai utilizar, aplicativos que necessitam ser instalados, comandos a serem executados, os volumes que serão montados, etc., etc., etc.!

É um *makefile* para criação de *containers*, e nele você passa todas as instruções para a criação do seu *container*.

Vamos ver como isso funciona na prática?

Primeira coisa: vamos criar um diretório onde deixaremos o nosso arquivo *dockerfile*, somente para ficar organizado. :D

Depois basta criar o *dockerfile* conforme exemplo a seguir:

```
# mkdir /root/primeiro_dockerfile
# cd /root/primeiro_dockerfile
# vim Dockerfile
```

Vamos adicionar as instruções que queremos para essa imagem de *container* que iremos criar:

```
FROM debian
RUN /bin/echo "HELLO DOCKER"
```

Apenas isso por enquanto. Salve e saia do *vim*.

Agora vamos rodar o comando “docker build” para fazer a criação dessa imagem de *container* utilizando o *dockerfile* criado.

```
root@linuxtips:~/primeiro_dockerfile# docker build .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM debian
latest: Pulling from library/debian

fdd5d7827f33: Pull complete
a3ed95caeb02: Pull complete
Digest:
sha256:e7d38b3517548a1c71e41bffe9c8ae6d6d29546ce46bf62159837aad072c90a
a
Status: Downloaded newer image for debian:latest
---> f50f9524513f
Step 2 : RUN /bin/echo "HELLO DOCKER"
---> Running in df60a0644bed
HELLO DOCKER
---> fd3af97a8940
Removing intermediate container df60a0644bed
Successfully built fd3af97a8940
root@linuxtips:~/primeiro_dockerfile#
```

Veja que usamos o diretório corrente, representado pelo caractere “.”, para indicar o *path* do meu arquivo *dockerfile*, mas você não precisa necessariamente estar no mesmo diretório, basta passar o *path* do diretório onde o arquivo se encontra.

Lembre apenas que é o *path* do diretório e não do arquivo.

É importante ressaltar que não passamos nem o nome nem a versão da imagem. Vamos aprender como fazê-lo mais adiante.

6.1. Outras opções

Vamos agora aprender um pouco mais sobre as opções que podemos utilizar quando estamos criando um *dockerfile*:

- ⇒ **ADD** – Copia novos arquivos, diretórios, arquivos TAR ou arquivos remotos e os adiciona ao *filesystem* do *container*.
- ⇒ **CMD** – Executa um comando. Diferentemente do RUN, que executa o comando no momento em que está *buildando* a imagem, o CMD o executa no início da execução do *container*.
- ⇒ **LABEL** – Adiciona metadados à imagem, como versão, descrição e fabricante.
- ⇒ **COPY** – Copia novos arquivos e diretórios e os adiciona ao *filesystem* do *container*.
- ⇒ **ENTRYPOINT** – Permite configurar um *container* para rodar um executável e quando esse executável for finalizado o *container* também será.
- ⇒ **ENV** – Informa variáveis de ambiente ao *container*.
- ⇒ **EXPOSE** – Informa em qual porta o *container* estará ouvindo.
- ⇒ **FROM** – Indica qual imagem será utilizada como base. Ela precisa ser a primeira linha do *dockerfile*.
- ⇒ **MAINTAINER** – Autor da imagem.
- ⇒ **RUN** – Executa qualquer comando em uma nova camada no topo da imagem e *commita* as alterações. Essas alterações você poderá utilizar nas próximas instruções de seu *dockerfile*.
- ⇒ **USER** – Determina qual o usuário que será utilizado na imagem. Por *default*, é o *root*.
- ⇒ **VOLUME** – Permite a criação de um ponto de montagem no *container*.
- ⇒ **WORKDIR** – Responsável por mudar do diretório / (raiz) para o especificado nele.

7. Entendendo volumes

7.1. Introdução a volumes no Docker

Bom, volumes nada mais são que diretórios (ou arquivos) que *bypassam* o *filesystem* do *container*. Decepcionei você? Que bom, sinal de que é bem simples e você não vai ter problemas para entender. :)

Volumes são especialmente designados para serem persistentes, inclusive independentemente do *container* onde eles são montados.

Existem algumas particularidades entre os volumes e *containers* que valem a pena ser mencionadas:

- ⇒ O volume é inicializado quando o *container* é criado.
- ⇒ Caso ocorra de já haver dados no diretório em que você está montando como volume, ou seja, se o diretório já existe e está “populado” na imagem base, aqueles dados serão copiados para o volume.
- ⇒ Um volume pode ser reusado e compartilhado entre *containers*.
- ⇒ Alterações em um volume são feitas diretamente no volume.
- ⇒ Alterações em um volume não irão com a imagem quando você fizer uma cópia ou *snapshot* de um *container*.
- ⇒ Volumes continuam a existir mesmo se você deletar o *container*.

Dito isso, chega de papo. Vamos aprender a adicionar um volume em um *container*.

```
root@linuxtips:~# docker run -ti -v /volume ubuntu /bin/bash
root@7db02e999bf2:/# df -h
```

```

Filesystem      Size  Used Avail Use% Mounted on
none            13G  6.8G      5.3G  57% /
tmpfs           999M      0   999M  0% /dev
tmpfs           999M      0   999M  0%
/sys/fs/cgroup
/dev/mapper/ubuntu--vg-root 13G  6.8G  5.3G  57% /volume
shm              64M      0   64M  0% /dev/shm
root@7db02e999bf2:/# ls
bin boot dev etc home lib lib64 media mnt opt proc
root run sbin srv sys tmp usr var volume
root@7db02e999bf2:/#

```

No exemplo anterior, conhecemos um novo parâmetro do comando “docker run”, o “-v”.

O parâmetro “-v” é o responsável por indicar o volume, que em nosso exemplo é o “/volume”. Perceba que, quando passamos o parâmetro “-v /volume”, o Docker montou esse diretório no *container*, porém sem nenhum conteúdo.

Podemos também montar um volume no *container* linkando-o com um diretório do *host*. Para exemplificar, vamos compartilhar o diretório “/root/primeiro_container”, que utilizamos para guardar o nosso primeiro *dockerfile*, e montá-lo no *container* em um volume chamado “/volume” da seguinte forma:

```

# docker run -ti -v /root/primeiro_dockerfile:/volume ubuntu
root@3d372a410ea2:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
none            13G  6.8G      5.3G  57% /
tmpfs           999M      0   999M  0% /dev
tmpfs           999M      0   999M  0%
/sys/fs/cgroup
/dev/mapper/ubuntu--vg-root 13G  6.8G  5.3G  57% /volume

```

```
shm                64M      0    64M  0% /dev/shm
```

```
root@3d372a410ea2:/#
```

Perceba que modificamos um pouco a forma de utilizar o “-v”; agora estamos passando da seguinte maneira:

```
-v /root/primeiro_dockerfile:/volume
-v diretorio_do_host:diretorio_do_container
```

Com isso, estamos montando o diretório “/root/primeiro_dockerfile” do *host* dentro do *container* com o nome de “/volume”.

No *container*:

```
root@3d372a410ea2:/# ls /volume/
Dockerfile
root@3d372a410ea2:/#
```

No *host*:

```
root@linuxtips:~# ls /root/primeiro_dockerfile/
Dockerfile
root@linuxtips:~#
```

Caso eu queira deixar o volume no *container* apenas como *read-only*, é possível. Basta passar o parâmetro “:ro” após o nome do volume:

```
# docker run -ti -v /root/primeiro_dockerfile:/volume:ro ubuntu
root@8d7863b1d9af:/# df -h
```

Filesystem	Size	Used	Avail	Use%	Mounted on
none	13G	6.8G	5.3G	57%	/
tmpfs	999M	0	999M	0%	/dev
tmpfs	999M	0	999M	0%	
/sys/fs/cgroup					
/dev/mapper/ubuntu--vg-root					
	13G	6.8G	5.3G	57%	/volume
shm	64M	0	64M	0%	/dev/shm

```
root@8d7863b1d9af:/# cd /volume/
root@8d7863b1d9af:/volume# ls
Dockerfile
root@8d7863b1d9af:/volume# mkdir teste
```

```
mkdir: cannot create directory 'teste': Read-only file system
root@8d7863b1d9af:/volume#
```

Assim como é possível montar um diretório como volume, também é possível montar um arquivo:

```
# docker run -ti -v
/root/primeiro_dockerfile/Dockerfile:/Dockerfile ubuntu
root@df0e3e58280a:/# df -h

Filesystem      Size  Used Avail Use% Mounted on
none            13G   6.8G   5.3G  57% /
tmpfs           999M    0    999M   0% /dev
tmpfs           999M    0    999M   0%
/sys/fs/cgroup
/dev/mapper/ubunt
--vg-root 13G  6.8G  5.3G  57%
/Dockerfile
shm            64M    0    64M   0% /dev/shm
root@df0e3e58280a:/# cat Dockerfile
FROM debian
RUN /bin/echo "HELLO DOCKER"
root@df0e3e58280a:/#
```

Isso faz com que o arquivo “/root/primeiro_dockerfile/Dockerfile” seja montado em “/Dockerfile” no *container*.

Simple como voar, não?

7.2. Localizando volumes

Caso você queira obter a localização do seu volume, é simples. Mas para isso você precisa conhecer um novo comando do Docker, o “docker inspect”. Nós até já demos uma pincelada com ele, mas não vimos a fundo. Vamos ver?

Com o “docker inspect” você consegue obter detalhes do seu *container*, como imagem base, pontos de montagem, configurações de rede, etc.

A saída do comando “docker inspect” é bastante extensa, por isso por enquanto vamos nos concentrar somente nas informações que ele traz em relação aos volumes.

```
# docker inspect -f {{.Mounts}} [CONTAINER ID]
[{{ /root/primeiro_dockerfile /volume ro false rprivate}}]
```

Como podemos notar, ele nos mostra o volume que está ativo no *container*, no caso o “/root/primeiro_dockerfile” do *host* montado no “/volume” do *container* como *read-only*, fácil assim!

O “{{.Mounts}}” é utilizado no Docker 1.8; no Docker 1.7 você utilizaria o “{{.Volumes}}”.

Caso você esteja usando Mac ou Windows, o Docker Machine monta o seu “/Users” (Mac) ou o seu “C:\Users” (Windows); portanto, esses diretórios serão montados normalmente no *container*. Qualquer coisa fora deles são diretórios do próprio Docker Machine.

7.3. Criando e montando um *data-only container*

Uma opção bastante interessante em relação aos volumes diz respeito ao *data-only container*, cuja única função é prover volumes para outros *containers*. Lembra do NFS *server* e do Samba? Ambos centralizavam diretórios com a finalidade de compartilhar entre outros servidores. Pois bem, o *data-only container* tem a mesma finalidade.

Um dos grandes baratos do Docker é a portabilidade. Um *container* criado no seu *laptop* deve ser portátil a ponto de rodar em qualquer outro ambiente que utilize o Docker, em todos os cantos do universo!

Sendo assim, o que acontece se eu criar um ambiente em Docker que diz para os *containers* montarem um diretório do *host* local? Depende. Depende de como está esse tal *host* local. Perguntas como “o diretório existe?” “as permissões estão ajustadas?”, entre outras mais, definirão o sucesso na execução dos *containers*, o que foge completamente do escopo do Docker, que tem como

principal característica a portabilidade. Se você possui um *cluster* com dez *hosts*, não é interessante usar diretório local, pois um *host* é diferente do outro.

Vamos ver como funciona isso na prática! :)

Para o nosso exemplo, primeiro vamos criar um *container* chamado “dbdados”, com um volume chamado “/data”, que guardará os dados do nosso banco PostgreSQL.

```
# docker create -v /data --name dbdados centos
```

Com isso, apenas criamos o *container* e especificamos um volume para ele, mas ainda não o iniciamos.

Sabemos que no *container* o volume se encontra montado em “/data”. Porém, qual a localização desse volume no *host*?

Lembra do “docker inspect”? Vamos utilizá-lo novamente:

```
root@linuxtips:~# docker inspect -f {{.Mounts}} dbdados
[{{46255137fe3f6d5f593e9ba9aaaf570b2f8b5c870f587c2fb34f29b79f9 7c30c
/var/lib/docker/volumes/46255137fe3f6d5f593e9ba9aaaf570b2f8b5c870f587c
2fb34f29b79f97c30c/_data /data local true }}]
```

Perceba que agora utilizamos o nome do *container* em vez do “CONTAINER ID”. Totalmente possível e muito mais intuitivo.

Quando executamos o “docker inspect”, ele nos retornou o caminho do nosso volume. Vamos ver se existe algum conteúdo dentro dele:

```
root@linuxtips:~# ls \
/var/lib/docker/volumes/46255137fe3f6d5f593e9ba9aaaf570b2f8b5c870f \
587c2fb34f29b79f97c30c/_data
```

Como vimos, o diretório ainda não possui conteúdo.

Agora vamos criar os *containers* que rodarão o PostgreSQL utilizando o volume “/data” do *container* “dbdados” para guardar os dados.

Para que possamos fazer o exemplo, precisamos conhecer mais dois parâmetros superimportantes:

⇒ **--volumes-from** – É utilizado quando queremos montar um volume disponibilizado por outro *container*.

⇒ **-e** – É utilizado para informar variáveis de ambiente para o *container*. No exemplo, estamos passando as variáveis de ambiente do PostgreSQL.

Pronto, agora estamos preparados! Vamos criar os *containers* com o PostgreSQL:

```
# docker run -d -p 5432:5432 --name pgsql1 --volumes-from dbdados \
-e POSTGRES_USER=docker -e POSTGRES_PASS=docker \
-e POSTGRES_DB=docker kamui/postgresql

# docker run -d -p 5433:5432 --name pgsql2 --volumes-from dbdados \
-e POSTGRES_USER=docker -e POSTGRES_PASS=docker \
-e POSTGRES_DB=docker kamui/postgresql
```

Para verificar os dois *containers* com o PostgreSQL em execução, utilize o “docker ps”.

Pronto, agora temos os dois *containers* com PostgreSQL em execução! Será que já temos algum dado no volume “/data” do *container* “dbdados”?

Vamos verificar novamente no *host* se o volume agora possui algum dado:

```
root@linuxtips:~# ls
/var/lib/docker/volumes/46255137fe3f6d5f593e9ba9aaaf570b2f8b5c870f587c
2fb34f29b79f97c30c/_data
base pg_clog pg_ident.conf pg_notify pg_snapshots
pg_stat_tmp pg_tblspc PG_VERSION postgresql.conf
postmaster.pid server.key global pg_hba.conf pg_multixact
pg_serial pg_stat pg_subtrans pg_twophase pg_xlog
postmaster.opts server.crt
root@linuxtips:~#
```

Sensacional! Como vimos, os dois *containers* do PostgreSQL estão escrevendo seus dados no volume “/data” do *container* “dbdados”. Chega a ser lacrimejante! :D

7.4. Sempre é bom ter um *backup*

Outra coisa bem bacana é a possibilidade de fazer *backups* dos seus *containers* de dados de forma muito simples e rápida.

Digamos que você queira fazer o *backup* do diretório “/data” do *container* “dbdados” que nós criamos há pouco; como faríamos?

```
root@linuxtips:~# cd backup/  
root@linuxtips:~/backup# docker run -ti --volumes-from dbdados -v  
$(pwd):/backup debian tar cvf /backup/backup.tar /data
```

Quando executamos o comando anterior, foi criado um novo *container* montando o(s) volume(s) do *container* “dbdados” (que no caso é o “/data”, lembra?). Além disso, será montado o diretório corrente do *host* no volume “/backup” do *container*, e em seguida será executado o comando do *tar* para empacotar o diretório “/data” dentro do diretório “/backup”.

```
root@linuxtips:~/backup# ls  
backup.tar  
root@linuxtips:~/backup#
```

8. Criando e gerenciando imagens

8.1. Agora eu quero criar minha imagem, posso?

Claro que pode!

E mais, vamos aprender de duas formas simples e intuitivas.

Uma das coisas mais interessantes do Docker é a possibilidade de usar imagens criadas por outras pessoas ao redor do mundo através de algum *registry* como o Docker Hub. Isso agiliza muito a sua vida, ainda mais quando você precisa apenas testar uma determinada tecnologia. O POC (*Proof of Concept* – em português, prova de conceito) se torna muito mais ágil, fazendo com que você consiga testar diversas ferramentas no mesmo tempo em que levaria para testar somente uma sem o Docker.

Entretanto, em determinados momentos precisamos criar a nossa própria imagem do zero, ou então modificar uma imagem criada por terceiros e salvar essas alterações em uma nova imagem.

Agora vamos ver os dois casos: como montar uma distribuição praticamente do zero utilizando somente instruções através do *dockerfile* e outra realizando modificações em uma imagem já existente e salvando em uma imagem nova.

8.2. Vamos começar do começo então, *dockerfile*!

Vamos montar a nossa primeira imagem utilizando como roteiro de criação um *dockerfile*. Você verá o quanto é simples a criação de um *dockerfile* bem

completo e prático. :)

Para começar, vamos criar um diretório chamado “/root/Dockerfiles”.

```
# mkdir /root/Dockerfiles
```

Agora começaremos a criação do nosso *dockerfile*, nosso mapa de criação da imagem. Para que possamos organizá-lo melhor, vamos criar um diretório chamado “apache”, onde guardaremos esse nosso primeiro exemplo:

```
# cd /root/Dockerfiles/  
# mkdir apache
```

Por enquanto, vamos apenas criar um arquivo chamado “Dockerfile” e adicionar o conteúdo conforme exemplo a seguir:

```
# cd apache  
# vim Dockerfile
```

```
FROM debian  
  
RUN apt-get update && apt-get install -y apache2 && apt-get clean  
ENV APACHE_LOCK_DIR="/var/lock"  
ENV APACHE_PID_FILE="/var/run/apache2.pid"  
ENV APACHE_RUN_USER="www-data"  
ENV APACHE_RUN_GROUP="www-data"  
ENV APACHE_LOG_DIR="/var/log/apache2"  
  
LABEL description="Webserver"  
  
VOLUME /var/www/html/  
EXPOSE 80
```

Após a criação do arquivo, vamos *buildar* (construir a nossa imagem) da seguinte forma:

```
# docker build .
```

Lembre-se: você deverá estar no diretório onde está o seu *dockerfile*.

Todos os passos que definimos em nosso *dockerfile* serão realizados, como a instalação dos pacotes solicitados e todas as demais tarefas.

```
Successfully built 53de2cee9e71
```

Muito bem! Como podemos notar na última linha da saída do “docker build”, a imagem foi criada com sucesso! :D

Vamos executar o “docker images” para ver se está tudo certo com a nossa primeira imagem!

```
root@linuxtips:~/Dockerfile/apache# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	53de2cee9e71	2 minutes ago	193.4 MB

A nossa imagem foi criada! Porém, temos um problema.

A imagem foi criada e está totalmente funcional, mas, quando a *buildamos*, não passamos o parâmetro “-t”, que é o responsável por adicionar uma *tag* (“nome:versão”) à imagem.

Vamos executar novamente o *build*, porém passando o parâmetro ‘-t’, conforme o exemplo a seguir:

```
# docker build -t linuxtips/apache:1.0 .
```

Agora vamos ver se realmente a imagem foi criada, adicionando um nome e uma versão a ela:

```
root@linuxtips:~/Dockerfile/apache# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
linuxtips/apache	1.0	53de2cee9e71	5 minutes ago	193.4 MB

Maravilha! Funcionou conforme esperávamos!

Vamos executar um *container* utilizando nossa imagem como base:

```
# docker run -ti linuxtips/apache:1.0
```

Agora já estamos no *container*. Vamos verificar se o Apache2 está em execução. Se ainda não estiver, vamos iniciá-lo e verificar se a porta 80 está “LISTEN”.

```
root@70dd36fe2d3b:/# ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	1	21:33	?	00:00:00	/bin/bash
root	6	1	0	21:33	?	00:00:00	ps -ef

```
root@70dd36fe2d3b:/# /etc/init.d/apache2 start
```

```
[....] Starting web server: apache2AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.4. Set the 'ServerName' directive globally to suppress this message
. ok
```

```
root@70dd36fe2d3b:/# ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	21:33	?	00:00:00	/bin/bash
root	30	1	0	21:33	?	00:00:00	
/usr/sbin/apache2 -k start							
www-data	33	30	0	21:33	?	00:00:00	
/usr/sbin/apache2 -k start							
www-data	34	30	0	21:33	?	00:00:00	
/usr/sbin/apache2 -k start							
root	109	1	0	21:33	?	00:00:00	ps -ef

```
root@70dd36fe2d3b:/# ss -atn
```

```
State Recv-Q Send-Q Local Address:Port Peer Address:Port
LISTEN 0 128 :::80 :::*
```

```
root@70dd36fe2d3b:/# ip addr show eth0
```

```
50: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.4/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:4/64 scope link
```



```
valid_lft forever preferred_lft forever
root@70dd36fe2d3b:/#
```

No código anterior é possível observar o IP do *container* na saída do “ip addr”. Vamos testar a comunicação com o *container* a partir do *host*.

No *host*, digite:

```
# curl <IP DO CONTAINER>
```

O “curl” retornou a página de boas-vindas do Apache2, ou seja, tudo está funcionando muito bem e o Apache2, respondendo conforme esperado!

8.3. Vamos customizar uma imagem base agora?

Vamos agora criar uma nova imagem, porém sem utilizar o *dockerfile*. Vamos executar um *container* com uma imagem base, realizar as modificações que desejarmos e depois salvar esse *container* como uma nova imagem!

Simples, rápido e fácil!

Bem, primeiro precisamos criar um *container*.

```
root@linuxtips:~# docker run -ti debian:8 /bin/bash
root@0b7e6f606aae:/#
```

Agora vamos fazer as alterações que desejamos. Vamos fazer o mesmo que fizemos quando montamos nossa primeira imagem com o *dockerfile*, ou seja, fazer a instalação do Apache2. :D

```
root@0b7e6f606aae:/# apt-get update && apt-get install -y apache2 &&
apt-get clean
```

Agora que já instalamos o Apache2, vamos sair do *container* para que possamos *commitar* a nossa imagem com base nesse *container* em execução:

Lembre-se de que para sair do *container* e deixá-lo ainda em execução é necessário pressionar **Ctrl + p + q**. ;)

```
# docker commit -m "meu container" CONTAINER ID
```

```
# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	fd131aedd43a	4 seconds ago	193.4 MB

Repare que nossa imagem ficou com o “<none>” em seu nome e “TAG”. Para que possamos ajustar e dar um nome e uma versão à nossa imagem, vamos usar o comando “docker tag”, conforme mostramos a seguir:

```
# docker tag IMAGEID linuxtips/apache_2:1.0
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
linuxtips/apache_2	1.0	fd131aedd43a	2 minutes ago	193.4 MB

Agora sim!!! Temos a nossa imagem criada e nome e versão especificados.

Vamos iniciar um *container* utilizando a imagem que acabamos de criar:

```
# docker run -ti linuxtips/apache_2:1.0 /bin/bash
```

Vamos subir o Apache2 e testar a comunicação do *container*:

```
root@57094ec894ce:/# ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	21:48	?	00:00:00	/bin/bash
root	6	1	0	21:48	?	00:00:00	ps -ef

```
root@57094ec894ce:/# /etc/init.d/apache2 start
```

```
[....] Starting web server: apache2AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.6. Set the 'ServerName' directive globally to suppress this message
. ok
```

```
root@70dd36fe2d3b:/# ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	21:43	?	00:00:00	/bin/bash
root	30	1	0	21:44	?	00:00:00	

```
/usr/sbin/apache2 -k start
```

www-data	33	30	0	21:44	?	00:00:00	
----------	----	----	---	-------	---	----------	--

```
/usr/sbin/apache2 -k start
```

```
www-data 34 30 0 21:44 ? 00:00:00
```

```
/usr/sbin/apache2 -k start
```

```
root 111 1 0 21:44 ? 00:00:00 ps -ef
```

```
root@70dd36fe2d3b:/# ss -atn
```

```
State Recv-Q Send-Q Local Address:Port Peer
```

```
Address:Port
```

```
LISTEN 0 128 :::80 :::*
```

```
root@57094ec894ce:/# ip addr show eth0
```

```
54: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue  
state UP group default
```

```
link/ether 02:42:ac:11:00:06 brd ff:ff:ff:ff:ff:ff
```

```
inet 172.17.0.6/16 scope global eth0
```

```
valid_lft forever preferred_lft forever
```

```
inet6 fe80::42:acff:fe11:6/64 scope link
```

```
valid_lft forever preferred_lft forever
```

Boaaa! Agora já temos o Apache2 em execução. Vamos sair do *container* e testar a comunicação com o Apache2 a partir do *host*:

```
# curl <Container IP>
```

Ele retornará a página de boas-vindas do Apache2! Tudo funcionando conforme esperado!

9. Compartilhando as imagens

Bem, já aprendemos como criar uma imagem, seja via *dockerfile* ou através da modificação de uma imagem base, e conhecemos alguns comandos interessantes, como o “docker build” e o “docker commit”.

Agora vamos aprender a compartilhar essas imagens, seja em um *registry* local ou então no *registry* do próprio Docker Hub.

9.1. O que é o Docker Hub?

Docker Hub é um repositório público e privado de imagens que disponibiliza diversos recursos, como, por exemplo, sistema de autenticação, *build* automático de imagens, gerenciamento de usuários e departamentos de sua organização, entre outras funcionalidades.

Pessoas e empresas se juntam, criam seus *containers* seguindo as melhores práticas, testam tudo direitinho e depois disponibilizam lá pra você usar sem ter nenhum trabalho. Isso é uma mão na roda gigantesca, uma vez que você não vai ter que perder tempo instalando coisas e às vezes até indo aprender como configurar tal serviço. Basta ir no Docker Hub e procurar; provavelmente alguém já criou um *container* que você pode usar pelo menos de base!

Provavelmente você não vai querer baixar da internet, mesmo que do *registry* do próprio Docker (sério), e subir no seu ambiente de produção algo que você não tem certeza de como funciona, o que é, etc.

Para resolver esse problema o Docker disponibiliza algumas funcionalidades, como o comando “docker inspect”, que já vimos antes, quando falávamos de volumes, lembra? Naquele momento usamos a *flag* “-f” e especificamos um campo de pesquisa, pois o intuito era mostrar somente a parte que estava sendo discutida naquele capítulo. Mas o “docker inspect” vai muito além disso; sem passar o “-f” ele vai retornar todas as informações contidas naquela imagem, desde a imagem base que foi utilizada até pontos de montagem, configurações, enfim, muita coisa. Teste aí:

```
root@linuxtips:~# docker inspect debian
[
  {
    "Id":
"sha256:f50f9524513f5356d952965dc97c7e831b02bb6ea0619da9bfc1997e4b9781
b7",
    "RepoTags": [
      "debian:8",
      "debian:latest"
    ],
    "RepoDigests": [],
    "Parent": "",
    "Comment": "",
    "Created": "2016-03-01T18:51:14.143360029Z",
    "Container":
"557177343b434b6797c19805d49c37728a4445d2610a6647c27055fbe4ec 3451",
    "ContainerConfig": {
      "Hostname": "e5c68db50333",
      "Domainname": "",
      "User": "",
      "AttachStdin": false,
      "AttachStdout": false,
      "AttachStderr": false,
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": null,
      "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) CMD [\"/bin/bash\"]"
      ],
    },
  ],
]
```

```

        "Image":
"d8bd0657b25f17eef81a3d52b53da5bda4de0cf5cca3dcafec277634ae4b 38fb",
        "Volumes": null,
        "WorkingDir": "",
        "Entrypoint": null,
        "OnBuild": null,
        "Labels": {}
    },
    "DockerVersion": "1.9.1",
    "Author": "",
    "Config": {
        "Hostname": "e5c68db50333",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": null,
        "Cmd": [
            "/bin/bash"
        ],
        "Image":
"d8bd0657b25f17eef81a3d52b53da5bda4de0cf5cca3dcafec277634ae4b 38fb",
        "Volumes": null,
        "WorkingDir": "",
        "Entrypoint": null,
        "OnBuild": null,
        "Labels": {}
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 125110803,
    "VirtualSize": 125110803,
    "GraphDriver": {
        "Name": "aufs",
        "Data": null
    }
}
]

```

root@linuxtips:~#

Às vezes será disponibilizado junto com a imagem o seu respectivo *dockerfile* e aí fica bem mais fácil: basta ler esse arquivo para saber exatamente como ela foi criada. :)

Um comando bastante interessante, que nos faz entender como uma imagem é dividida em camadas (e, principalmente, o que foi feito em cada camada), é o “docker history”.

```
root@linuxtips:~# docker history linuxtips/apache:1.0
IMAGE                CREATED              CREATED BY          SIZE  COMMENT
4862def18dfd        36 minutes ago     /bin/sh -c #(nop)  EXPOSE
80/tcp 0 B
06210ac863da        36 minutes ago     /bin/sh -c #(nop)  VOLUME
[/var/www/html/] 0 B
fed9b6bc7ad9        36 minutes ago     /bin/sh -c #(nop)  LABEL
description=Webserver 0 B
68f6e8de3df3        36 minutes ago     /bin/sh -c #(nop)  ENV
APACHE_LOG_DIR=/var/log 0 B
1a129e753d1e        36 minutes ago     /bin/sh -c #(nop)  ENV
APACHE_RUN_GROUP=www-da 0 B
f0f9d7be7c70        36 minutes ago     /bin/sh -c #(nop)  ENV
APACHE_RUN_USER=www-dat 0 B
3dafea4a403a        36 minutes ago     /bin/sh -c #(nop)  ENV
APACHE_PID_FILE=/var/ru 0 B
f31eb176ecc8        36 minutes ago     /bin/sh -c #(nop)  ENV
APACHE_LOCK_DIR=/var/lo 0 B
0bbefd91da05        36 minutes ago     /bin/sh -c apt-get update &&
apt-get install 68.29 MB
f50f9524513f        12 days ago        /bin/sh -c #(nop)  CMD
["/bin/bash"]      0 B
```

```
<missing>      12 days ago      /bin/sh -c #(nop) ADD
file:b5393172fb513d 125.1 MB
root@linuxtips:~#
```

Perceba que as primeiras linhas da saída do comando anterior são referentes às informações que pedimos para adicionar à imagem no *dockerfile*. As demais camadas são originais da imagem que pegamos do Docker Hub através da instrução “FROM”.

Existe também um site chamado “ImageLayers”: ele faz exatamente a mesma coisa que o “docker history”, mas você não precisa baixar a imagem – e, bom, é *web*. O ImageLayers pode ser acessado em: <https://imagelayers.io/>.

O Docker Hub, como já falamos, tem muitos componentes, dentre eles o responsável pelo repositório de imagens: o *registry*.

É possível utilizar um *registry* local em vez de um na nuvem, como o Docker Hub ou outros *registries* que são facilmente encontrados na internet. Falaremos disso com detalhes mais à frente. :P

Para que você possa utilizar o Docker Hub para gerenciar as suas imagens, é necessário criar uma conta. Existem duas formas de fazer isso: através da URL <http://hub.docker.com/account/signup> ou então através da linha de comando utilizando o “docker login”.

9.2. Vamos criar uma conta?

Vamos criar uma conta no Docker Hub, porém não iremos fazer através do navegador; vamos criar via linha de comando, pois acreditamos ser esta a maneira mais eficiente e fácil de realizar essa tarefa.

Para que consiga realizar a criação da sua conta, é necessário utilizar o comando “docker login”. Ele também é utilizado para autenticar no Docker Hub após a criação da conta.

Vamos criar uma conta como exemplo:


```
root@linuxtips:~# docker login
Username: linuxtips
Password:
Email: linuxtipsbr@gmail.com
WARNING: login credentials saved in /root/.docker/config.json
Account created. Please use the confirmation link we sent to your e-
mail to activate it.
root@linuxtips:~#
```

Após inserir usuário, senha e e-mail, é enviada uma mensagem de confirmação para o endereço que você informou.

A partir da confirmação do seu e-mail, já é possível se autenticar e começar a fazer uso do Docker Hub.

```
root@linuxtips:~# docker login
Username: linuxtips
Password:
Email: linuxtipsbr@gmail.com
WARNING: login credentials saved in /root/.docker/config.json
Login Succeeded
root@linuxtips:~#
```

Você consegue criar repositórios públicos à vontade, porém na conta *free* você somente tem direito a um repositório privado. Caso precise de mais do que um repositório privado, é necessário o *upgrade* da sua conta e o pagamento de uma mensalidade. :)

Caso você queira especificar outro *registry* em vez do Docker Hub, basta passar o endereço como parâmetro, como segue:

```
# docker login registry.seilaqual.com
```

9.3. Agora vamos compartilhar essas imagens na *interwebs*!

Uma vez que já criamos a nossa conta no Docker Hub, podemos começar a utilizá-la!

Como exemplo, vamos utilizar a imagem que montamos com o *dockerfile* no capítulo anterior chamada “linuxtips/apache”. Quando realizarmos o *upload* dessa imagem para o Docker Hub, o repositório terá o mesmo nome da imagem, ou seja, “linuxtips/apache”.

Uma coisa muito importante! A sua imagem deverá ter o seguinte padrão, para que você consiga fazer o *upload* para o Docker Hub:

seuusuario/nomedaimagem:versão

Assim, sabemos que “linuxtips/apache:1.0” significa:

⇒ **linuxtips** – Usuário do Docker Hub.

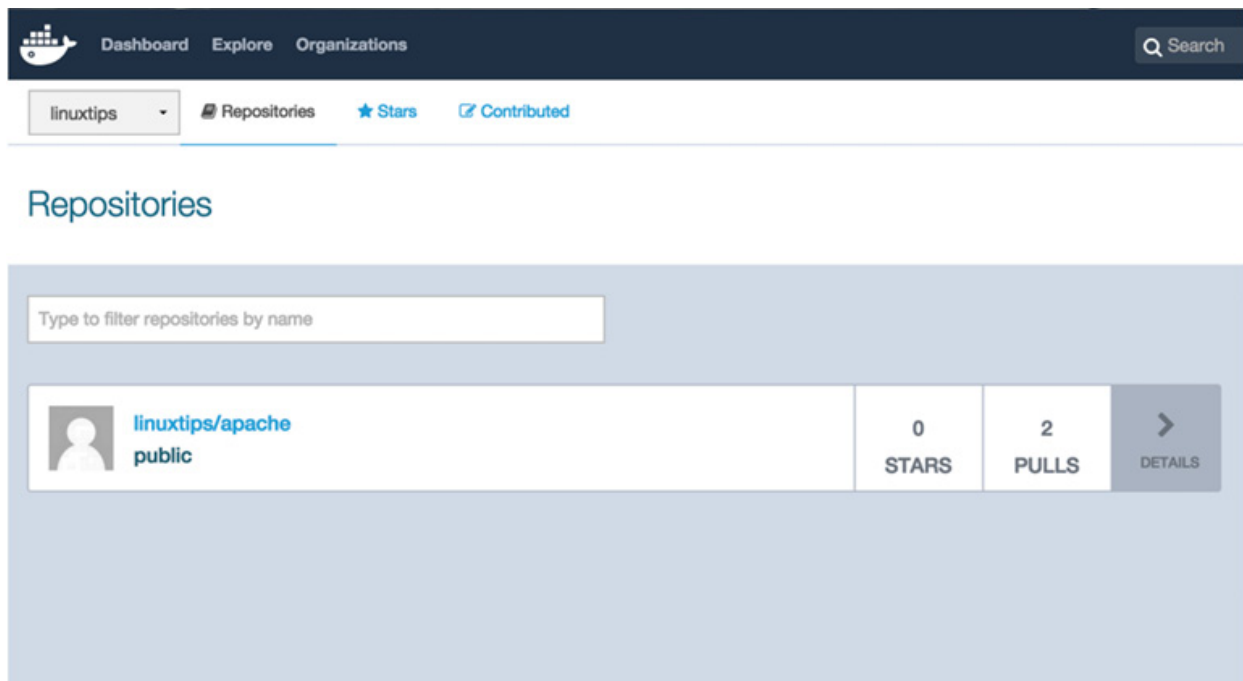
⇒ **apache** – Nome da imagem.

⇒ **1.0** – Versão.

Agora vamos utilizar o comando “docker push”, responsável por fazer o *upload* da imagem da sua máquina local para o *registry* do Docker Hub, como mostrado no exemplo a seguir:

```
root@linuxtips:~# docker push linuxtips/apache:1.0
The push refers to a repository [docker.io/linuxtips/apache]
b3a691489ee1: Pushed
5f70bf18a086: Layer already exists
917c0fc99b35: Pushed
1.0: digest:
sha256:c8626093b19a686fd260dbe0c12db79a97ddfb6a6d8e4c4f44634f66991d93d
0 size: 6861
root@linuxtips:~#
```

Acessando a URL <https://hub.docker.com/> você conseguirá visualizar o repositório que acabou de criar, conforme a imagem a seguir:



Por padrão, ele cria o repositório como público. ;)

Caso você queira visualizar o seu novo repositório pela linha de comando, basta utilizar o comando “docker search” seguido de seu usuário do Docker Hub:

```
root@linuxtips:~# docker search <seu_usuario>
NAME                                DESCRIPTION STARS  OFFICIAL
AUTOMATED
linuxtips/apache                    0
```

Já que possuímos a imagem no *registry* do Docker Hub, vamos testá-la fazendo o *pull* da imagem e em seguida vamos executando-a para saber se realmente tudo isso funciona de forma simples e fácil assim. :)

Primeiro vamos parar os *containers* que utilizam a imagem “linuxtips/apache”:

```
# docker ps | grep seu_usuario/sua_imagem
# docker stop CONTAINERID
```

Não é possível remover uma imagem se algum *container* estiver em execução utilizando-a como imagem base. Por isso é necessário parar os *containers* conforme fizemos antes.

Para que você possa remover uma imagem é necessário utilizar o comando “docker rmi”, responsável por remover imagens do disco local.

É importante mencionar que, caso possua *containers* parados que utilizam essa imagem como base, é necessário forçar a remoção da imagem utilizando o parâmetro “-f”:

```
# docker rmi -f linuxtips/apache:1.0
Untagged: linuxtips/apache:1.0
```

Pronto! Removemos a imagem!

Agora vamos realizar o *pull* da imagem diretamente do *registry* do Docker Hub para que tenhamos a imagem novamente em nosso disco local e assim possamos subir um *container* utilizando-a.

```
root@linuxtips:~# docker pull linuxtips/apache:1.0
1.0: Pulling from linuxtips/apache
fdd5d7827f33: Already exists
a3ed95caeb02: Already exists
11b590220174: Already exists
Digest:
sha256:c8626093b19a686fd260dbe0c12db79a97ddfb6a6d8e4c4f44634f66991d93d
0
Status: Downloaded newer image for linuxtips/apache:1.0
root@linuxtips:~#
```

Podemos novamente visualizá-la através do comando “docker images”.

```
root@linuxtips:~# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
linuxtips/apache	1.0	4862def18dfd	About an hour ago

```
193.4 MB
root@linuxtips:~#
```

Para criar o *container* utilizando nossa imagem:

```
root@linuxtips:~# docker run -ti linuxtips/apache:1.0  
/bin/bash
```

Simples como voar, não?

9.4. Não confio na internet; posso criar o meu *registry* local?

Como algumas empresas não gostam de manter seus dados na nuvem em serviços de terceiros, existe a possibilidade de você configurar um *registry* local. Assim, você não precisa utilizar o *registry* do Docker Hub, por exemplo. Isso permite a você compartilhar suas imagens com outras pessoas de sua empresa, funcionando como repositório de imagens Docker. Sensacional!

A URL do projeto fica em <https://github.com/docker/distribution>. O Docker Distribution é um *registry* que serve para guardar e compartilhar suas imagens. Ele substitui o Docker Registry, que se tornou obsoleto.

Para que possamos ter o Docker Distribution de forma simples e totalmente funcional, guardando e distribuindo nossas imagens Docker localmente, basta rodá-lo como um *container*! :D

```
root@linuxtips:~# docker run -d -p 5000:5000 --restart=always --name  
registry registry:2  
Unable to find image 'registry:2' locally  
2: Pulling from library/registry  
fdd5d7827f33: Already exists  
a3ed95caeb02: Pull complete  
a79b4a92697e: Pull complete  
6cbb75c7cc30: Pull complete  
4831699594bc: Pull complete  
Digest:  
sha256:20f5d95004b71fe14dbe7468eff33f18ee7fa52502423c5d107d4fb0abb05c1  
d  
Status: Downloaded newer image for registry:2  
4f8efc8a71531656dc74e99dea74da203645c0f342b0706bc74200ae0a50cb20  
root@linuxtips:~#
```

Com o comando anterior, criamos um *container* chamado “registry” que utiliza a imagem “registry:2” como base e usamos a opção “--restart=always”. Caso ocorra qualquer problema com o *container* ou com o Docker, ele será iniciado automaticamente. Passamos também que a porta de comunicação com o *container* será a 5000, que também utilizará a porta 5000 do *host* com o mesmo propósito. Vamos ver sobre o parâmetro “-p” em breve, no capítulo relacionado a redes. ;)

Você pode verificar o *container* do *registry* em execução, bem como sua imagem:

```
# docker ps
# docker images
```

Muito bom, nosso *registry* já está em execução! Agora vamos testá-lo tentando realizar um *push* da nossa imagem para ele.

Primeiramente, teremos que adicionar uma nova *tag* em nossa imagem mencionando o endereço do novo *registry* em vez do nome do usuário que utilizávamos quando queríamos fazer o *push* para o Docker Hub. Para isso, vamos utilizar novamente o comando “docker tag”:

```
# docker tag IMAGEID localhost:5000/apache
# docker images
```

Agora basta fazer o *push* para o nosso *registry* local da seguinte forma:

```
root@linuxtips:~# docker push localhost:5000/apache
The push refers to a repository [localhost:5000/apache]
b3a691489ee1: Pushed
5f70bf18a086: Pushed
917c0fc99b35: Pushed
latest:
sha256:0e69b8d5cea67fcfedb5d7128a9fd77270461aa5852e6fe9b465565ec8e4e12
f size: 925
root@linuxtips:~#
```

Pronto! Agora nós possuímos um *registry* local!

Fizemos um *registry* totalmente funcional, porém simples. Caso queira utilizar recursos como controle de usuários, certificados, outras opções de *storage*, etc., visite [a URL: https://github.com/docker/distribution/blob/master/docs/deploying.md](https://github.com/docker/distribution/blob/master/docs/deploying.md).

10. Gerenciando a rede dos *containers*

Quando o Docker é executado, ele cria uma *bridge* virtual chamada “docker0”, para que possa gerenciar a comunicação interna entre o *container* e o *host* e também entre os *containers*.

Vamos conhecer alguns parâmetros do comando “docker run” que irão nos ajudar com a rede onde os *containers* irão se comunicar.

⇒ **--dns** – Indica o servidor DNS.

⇒ **--hostname** – Indica um *hostname*.

⇒ **--link** – Cria um *link* entre os *containers*, sem a necessidade de se saber o IP um do outro.

⇒ **--net** – Permite configurar o modo de rede que você usará com o *container*. Temos quatro opções, mas a mais conhecida e utilizada é a “--net=host”, que permite que o *container* utilize a rede do *host* para se comunicar e não crie um *stack* de rede para o *container*.

⇒ **--expose** – Expõe a porta do *container* apenas.

⇒ **--publish** – Expõe a porta do *container* e do *host*.

⇒ **--default-gateway** – Determina a rota padrão.

⇒ **--mac-address** – Determina um MAC *address*.

Quando o *container* é iniciado, a rede passa por algumas etapas até a sua inicialização completa:

1. Cria-se um par de interfaces virtuais.
2. Cria-se uma interface com nome único, como “veth1234”, e em seguida *linka-se* com a *bridge* do Docker, a “docker0”.
3. Com isso, é disponibilizada a interface “eth0” dentro do *container*, em um *network namespace* único.
4. Configura-se o MAC *address* da interface virtual do *container*.
5. Aloca-se um IP na “eth0” do *container*. Esse IP tem que pertencer ao *range* da *bridge* “docker0”.

Com isso, o *container* já possui uma interface de rede e já está apto a se comunicar com outros *containers* ou com o *host*. :D

10.1. Consigo fazer com que a porta do *container* responda na porta do *host*?

Sim, isso é possível e bastante utilizado.

Vamos conhecer um pouco mais sobre isso em um exemplo utilizando aquela nossa imagem “linuxtips/apache”.

Primeira coisa que temos que saber é a porta onde o Apache2 se comunica. Isso é fácil, né? Se estiver com as configurações padrões de porta de um *web server*, o Apache2 do *container* estará respondendo na porta 80/TCP, correto?

Agora vamos fazer com que a porta 8080 do nosso *host* responda pela porta 80 do nosso *container*, ou seja, sempre que alguém bater na porta 8080 do nosso *host*, a requisição será encaminhada para a porta 80 do *container*. Simples, né?

Para conseguir fazer esse encaminhamento, precisamos utilizar o parâmetro “-p” do comando “docker run”, conforme faremos no exemplo a seguir:

```
root@linuxtips:~# # docker run -ti -p 8080:80
linuxtips/apache:1.0 /bin/bash
root@4a0645de6d94:/# ps -ef
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	1	18:18	?	00:00:00	/bin/bash
root	6	1	0	18:18	?	00:00:00	ps -ef

root@4a0645de6d94:/# /etc/init.d/apache2 start

[....] Starting web server: apache2AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.3. Set the 'ServerName' directive globally to suppress this message
. ok

root@4a0645de6d94:/# ps -ef

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	18:18	?	00:00:00	/bin/bash
root	30	1	0	18:19	?	00:00:00	
/usr/sbin/apache2 -k start							
www-data	33	30	0	18:19	?	00:00:00	
/usr/sbin/apache2 -k start							
www-data	34	30	0	18:19	?	00:00:00	
/usr/sbin/apache2 -k start							
root	109	1	0	18:19	?	00:00:00	ps -ef

root@4a0645de6d94:/# ip addr

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
74: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link
        valid_lft forever preferred_lft forever
```

root@4a0645de6d94:/#

Repare que passamos o parâmetro “-p” da seguinte forma:

⇒ -p 8080:80 – Onde “8080” é a porta do *host* e “80”, a do *container*.

Com isso, estamos dizendo que toda requisição que chegar na porta 8080 do meu *host* deverá ser encaminhada para a porta 80 do *container*.

Já no *container*, subimos o Apache2 e verificamos o IP do *container*, correto?

Agora vamos sair do *container* com o atalho “Ctrl + p + q”. :)

A partir do *host*, vamos realizar um “curl” com destino ao IP do *container* na porta 80, depois com destino à porta 8080 do *host* e em seguida analisar as saídas:

```
root@linuxtips:~# curl <IPCONTAINER>:80
```

Se tudo ocorreu bem até aqui, você verá o código da página de boas-vindas do Apache2.

O mesmo ocorre quando executamos o “curl” novamente, porém batendo no IP do *host*. Veja:

```
root@linuxtips:~# curl <IPHOST>:8080
```

Muito fácil, chega a ser lacrimejante! \o/

10.2. E como ele faz isso? Mágica?

Não, não é mágica! Na verdade, o comando apenas utiliza um módulo bastante antigo do *kernel* do Linux chamado *netfilter*, que disponibiliza a ferramenta *iptables*, que todos nós já cansamos de usar.

Vamos dar uma olhada nas regras de *iptables* referentes a esse nosso *container*. Primeiro a tabela *filter*:

```
root@linuxtips:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target prot opt source destination
```

```
Chain FORWARD (policy ACCEPT)
target prot opt source destination
DOCKER-ISOLATION all -- 0.0.0.0/0 0.0.0.0/0
DOCKER all -- 0.0.0.0/0 0.0.0.0/0
ACCEPT all -- 0.0.0.0/0 0.0.0.0/0
ctstate RELATED,ESTABLISHED
ACCEPT all -- 0.0.0.0/0 0.0.0.0/0
ACCEPT all -- 0.0.0.0/0 0.0.0.0/0
```

```
Chain OUTPUT (policy ACCEPT)
target prot opt source destination
```

```
Chain DOCKER (1 references)
target prot opt source destination
ACCEPT tcp -- 0.0.0.0/0 172.17.0.2
tcp dpt:5000
ACCEPT tcp -- 0.0.0.0/0 172.17.0.3
tcp dpt:80
```

```
Chain DOCKER-ISOLATION (1 references)
target prot opt source destination
RETURN all -- 0.0.0.0/0 0.0.0.0/0
```

root@linuxtips:~#

Agora a tabela NAT:

root@linuxtips:~# iptables -L -n -t nat

```
Chain PREROUTING (policy ACCEPT)
target prot opt source destination
DOCKER all -- 0.0.0.0/0 0.0.0.0/0
ADDRTYPE match dst-type LOCAL
```

```
Chain INPUT (policy ACCEPT)
target prot opt source destination
```

```

Chain OUTPUT (policy ACCEPT)
  target prot opt source destination
  DOCKER all -- 0.0.0.0/0 !127.0.0.0/8
  ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT)
  target      prot opt source      destination
  MASQUERADE  all  -- 172.17.0.0/16 0.0.0.0/0
  MASQUERADE  tcp  -- 172.17.0.2    172.17.0.2
  tcp dpt:5000
  MASQUERADE  tcp  -- 172.17.0.3    172.17.0.3
  tcp dpt:80

Chain DOCKER (2 references)
  target prot opt source destination
  RETURN all -- 0.0.0.0/0 0.0.0.0/0
  DNAT   tcp  -- 0.0.0.0/0 0.0.0.0/0
  tcp dpt:5000 to:172.17.0.2:5000
  DNAT   tcp  -- 0.0.0.0/0 0.0.0.0/0
  tcp dpt:8080 to:172.17.0.3:80
root@linuxtips:~#

```

Como podemos notar, temos regras de NAT configuradas que permitem o DNAT da porta 8080 do *host* para a 80 do *container*. Veja a seguir:

```

MASQUERADE tcp -- 172.17.0.3 172.17.0.3
tcp dpt:80
DNAT      tcp -- 0.0.0.0/0 0.0.0.0/0
tcp dpt:8080 to:172.17.0.3:80

```

Tudo isso feito “automagicamente” pelo Docker, sem a necessidade de precisar configurar diversas regras de *iptables*. <3

| 1. Controlando o *daemon* do Docker

Antes de tudo, vamos tentar entender o que é um *daemon*. Sabemos que, em sistemas operacionais *multitask*, isto é, em um sistema operacional capaz de executar mais de uma tarefa por vez (*not really*), um *daemon* é um software que roda de forma independente em *background*. Ele executa certas ações predefinidas em resposta a certos eventos. Pois bem, o *daemon* do Docker é exatamente isso: uma espécie de processo-pai que controla tudo, *containers*, imagens, etc., etc., etc.

Até o Docker 1.7 as configurações referentes especificamente ao *daemon* se confundiam bastante com configurações globais – isso porque quando você digitava lá o “docker -help” um monte de coisas retornava, e você não sabia o que era o quê. A partir da versão 1.8 foi introduzido um novo comando, o “docker daemon”, que resolve de vez esse problema e trata especificamente de configurações referentes, obviamente, ao *daemon* do Docker.

11.1. O Docker sempre utiliza 172.16.X.X ou posso configurar outro intervalo de IP?

Sim, você pode configurar outro *range* para serem utilizados pela *bridge* “docker0” e também pelas interfaces dos *containers*.

Para que você consiga configurar um *range* diferente para utilização do Docker é necessário iniciá-lo com o parâmetro “--bip”.

```
# docker daemon --bip 192.168.0.1/24
```

Assim, você estará informando ao Docker que deseja utilizar o IP “192.168.0.1” para sua *bridge* “docker0” e, conseqüentemente, para a *subnet* dos *containers*.

Você também poderá utilizar o parâmetro “--fixed-cidr” para restringir o *range* que o Docker irá utilizar para a *bridge* “docker0” e para a *subnet* dos *containers*.

```
# docker daemon --fixed-cidr 192.168.0.0/24
```

11.2. Opções de *sockets*

Sockets são *end-points* onde duas ou mais aplicações ou processos se comunicam em um ambiente, geralmente um “IP:porta” ou um arquivo, como no caso do *Unix Domain Sockets*.

Atualmente o Docker consegue trabalhar com três tipos de *sockets*, Unix, TCP e FD, e por *default* ele usa *unix sockets*. Você deve ter notado que, ao *startar* seu Docker, foi criado um arquivo em “/var/run/docker.sock”. Para fazer alterações nele você vai precisar ou de permissão de *root* ou de que o usuário que está executando as ações esteja no grupo “docker”, como dissemos no começo deste livro, lembra?

Por mais prático que isso seja, existem algumas limitações, como, por exemplo, o *daemon* só poder ser acessado localmente. Para resolver isso usamos geralmente o TCP. Nesse modelo nós definimos um IP, que pode ser tanto “qualquer um” (0.0.0.0 e uma porta) como um IP específico e uma porta.

Nos sistemas baseados em *systemd* você ainda pode se beneficiar do *systemd socket activation*, uma tecnologia que visa economia de recursos. Consiste basicamente em ativar um *socket* somente enquanto uma conexão nova chega e desativar quando não está sendo mais usado.

Além disso tudo, dependendo do seu ambiente, você também pode fazer o Docker escutar em diferentes tipos de *sockets*, o que é conseguido através do parâmetro “-H” do comando “docker daemon”.

Exemplos:

11.2.1. *Unix Domain Socket*

```
root@linuxtips:~# docker daemon -H
unix:///var/run/docker.sock
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.00 seconds
INFO[0000] Firewalld running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.0/16. Daemon option --bip can be used to set a preferred IP
address
WARN[0000] Your kernel does not support swap memory limit.
INFO[0000] Loading containers: start.
.....
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=c3959b1 execdriver=native-0.2 graphdriver=aufs
version=1.10.2
INFO[0000] API listen on /var/run/docker.sock
```

11.2.2. TCP

```
root@linuxtips:~# docker daemon -H tcp://0.0.0.0:2375
WARN[0000] /!\ DON'T BIND ON ANY IP ADDRESS WITHOUT setting -tlsverify
IF YOU DON'T KNOW WHAT YOU'RE DOING /!\
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.01 seconds
INFO[0000] Firewalld running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.0/16. Daemon option --bip can be used to set a preferred IP
address
WARN[0000] Your kernel does not support swap memory limit.
INFO[0000] Loading containers: start.
.....
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=c3959b1 execdriver=native-0.2 graphdriver=aufs
version=1.10.2
INFO[0000] API listen on [::]:2375
```

11.3. Opções de *storage*

Como falamos anteriormente, o Docker suporta alguns *storage drivers*, todos baseados no esquema de *layers*. Essas opções são passadas para o *daemon* pelo parâmetro “--storage-opt”, onde itens relacionados ao *Device Mapper* recebem o prefixo “dm” e “zfs” para (adivinha?) o ZFS. A seguir vamos demonstrar algumas opções mais comuns:

⇒ **dm.thinpooldev** – Com esta opção você consegue especificar o *device* que será usado pelo *Device Mapper* para desenvolver o *thin-pool* que ele usa para criar os *snapshots* usados por *containers* e imagens.

Exemplo:

```
root@linuxtips:~# docker daemon --storage-opt dm.thinpooldev=/dev/mapper/thin-pool
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.00 seconds
INFO[0000] Firewall running: false
INFO[0000] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. Daemon option --bip can be used to set a preferred IP address
WARN[0000] Your kernel does not support swap memory limit.
INFO[0000] Loading containers: start.
.....
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=c3959b1 execdriver=native-0.2 graphdriver=aufs
version=1.10.2
INFO[0000] API listen on /var/run/docker.sock
```

⇒ **dm.basesize** – Este parâmetro define o tamanho máximo do *container*. O chato disso é que você precisa deletar tudo dentro de “/var/lib/docker” (o que implica em matar todos os *containers* e imagens) e *restartar* o serviço do Docker.

```
root@linuxtips:~# docker daemon --storage-opt dm.basesize=10G
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] Graph migration to content-addressability took 0.00 seconds
INFO[0000] Firewall running: false
```

```
INFO[0000] Default bridge (docker0) is assigned with an IP address
172.17.0.0/16. Daemon option --bip can be used to set a preferred IP
address
WARN[0000] Your kernel does not support swap memory limit.
INFO[0000] Loading containers: start.
.....
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=c3959b1 execdriver=native-0.2 graphdriver=aufs
version=1.10.2
INFO[0000] API listen on /var/run/docker.sock
```

⇒ **dm.fs** – Especifica o *filesystem* do *container*. As opções suportadas são: **EXT4** e **XFS**.

11.4. Opções diversas

- ⇒ **--default-ulimit** – Passando isso para o *daemon*, todos os *containers* serão *startados* com esse valor para o “ulimit”. Esta opção é sobrescrita pelo parâmetro “--ulimit” do comando “docker run”, que geralmente vai dar uma visão mais específica.
- ⇒ **--icc** – “icc” vem de *inter container communication*. Por padrão, ele vem marcado como *true*; caso você não queira esse tipo de comunicação, você pode marcar no *daemon* como *false*.
- ⇒ **--log-level** – É possível alterar também a forma como o Docker trabalha com *log*; em algumas situações (geralmente *troubleshoot*) você pode precisar de um *log* mais “verboso”, por exemplo.

12. Utilizando Docker Machine, Docker Swarm e Docker Compose

Chegamos no momento em que vamos começar a brincar de maestro!

Agora vamos começar a utilizar os *containers* em maior volume. Vamos aprender como fazer para controlar um grande número de *containers* de forma simples e utilizando somente as ferramentas que o Docker nos disponibiliza.

A seguir trataremos de três componentes muito importantes do ecossistema do Docker: o Docker Machine, o Docker Swarm e o Docker Compose!

12.1. Ouvi dizer que minha vida ficaria melhor com o Docker Machine!

Certamente!

Com o Docker Machine você consegue, com apenas um comando, iniciar o seu projeto com Docker!

Antes do Docker Machine, caso quiséssemos montar um Docker Host, era necessário fazer a instalação do sistema operacional, instalar e configurar o Docker e outras ferramentas que se fazem necessárias.

Perderíamos um tempo valioso com esses passos, sendo que já poderíamos estar trabalhando efetivamente com o Docker e seus *containers*.

Porém, tudo mudou com o Docker Machine! Com ele você consegue criar o seu Docker Host com apenas um comando. O Docker Machine consegue

trabalhar com os principais *hypervisors* de VMs, como o VMware, Hyper-V e Oracle VirtualBox, e também com os principais provedores de infraestrutura, como AWS, Google Compute Engine, DigitalOcean, Rackspace, Azure, etc.

Aqui tem a lista completa de todos os *drivers* que o Docker Machine suporta: <https://docs.docker.com/machine/drivers/>.

Quando você utiliza o Docker Machine para instalar um Docker Host na AWS, por exemplo, ele disponibilizará uma máquina com Linux e com o Docker e suas dependências já instaladas.

12.1.1. Vamos instalar?

A instalação do Docker Machine é bastante simples. Vale lembrar que é possível instalar o Docker Machine no Linux, MacOS ou Windows.

Para fazer a instalação do Docker Machine no Linux, faça:

```
# wget  
https://github.com/docker/machine/releases/download/v0.6.0/docker-  
machine-linux-x86_64  
  
# mv docker-machine-linux-x86_64 /usr/local/bin/docker-machine  
  
# chmod +x /usr/local/bin/docker-machine
```

Para verificar se ele foi instalado e qual a sua versão, faça:

```
root@linuxtips:~# docker-machine --version  
docker-machine version 0.6.0, build e27fb87  
root@linuxtips:~#
```

Pronto. Como tudo que é feito pelo Docker, é simples de instalar e fácil de operar. :)

12.1.2. Vamos iniciar nosso primeiro projeto?

Agora que já temos o Docker Machine instalado em nossa máquina, já conseguiremos fazer a instalação do Docker Host de forma bastante simples –

lembrando que mesmo que tivéssemos feito a instalação do Docker Machine no Windows, conseguiríamos tranquilamente comandar a instalação do Docker Hosts na AWS em máquinas Linux. Tenha em mente que a máquina onde você instalou o Docker Machine é o maestro que determina a criação de novos Docker Hosts, seja em VMs ou em alguma nuvem como a AWS.

Em nosso primeiro projeto, vamos fazer com que o Docker Machine faça a instalação do Docker Host utilizando o VirtualBox.

Como iremos utilizar o VirtualBox, é evidente que precisamos ter instalado o VirtualBox em nossa máquina para que tudo funcione. ;)

Portanto:

```
root@linuxtips:~# apt-get install virtualbox
```

Para fazer a instalação de um novo Docker Host, utilizamos o comando “docker-machine create”. Para escolher onde iremos criar o Docker Host, utilizamos o parâmetro “--driver”, conforme segue:

```
root@linuxtips:~# docker-machine create --driver virtualbox linuxtips
Running pre-create checks...
Creating machine...
Waiting for machine to be running, this may take a few
minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Provisioning created instance...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
To see how to connect Docker to this machine, run:
docker-machine env linuxtips
root@linuxtips:~#
```

Onde:

⇒ **docker-machine create** – Cria um novo Docker Host.

⇒ **--driver virtualbox** – Irá criá-lo utilizando o VirtualBox.

⇒ **linuxtips** – Nome da VM que será criada.

Para visualizar o *host* que acabou de criar, basta digitar o seguinte comando:

```
root@linuxtips:~# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM
linuxtips	-	virtualbox	Running		

```
tcp://192.168.99.100:2376
```

Como podemos notar, o nosso *host* está sendo executado perfeitamente! Repare que temos uma coluna chamada URL, correto? Nela temos a URL para que possamos nos comunicar com o nosso novo *host*.

Outra forma de visualizar informações sobre o *host*, mais especificamente sobre as variáveis de ambiente dele, é digitar:

```
root@linuxtips:~# docker-machine env linuxtips
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/jvitali/.docker/machine/
machines/linuxtips"
export DOCKER_MACHINE_NAME="linuxtips"
# Run this command to configure your shell:
# eval "$(docker-machine env linuxtips)"
root@linuxtips:~#
```

Serão mostradas todas as variáveis de ambiente do *host*, como URL, certificado e nome.

Para que você possa acessar o ambiente desse *host* que acabamos de criar, faça:

```
root@linuxtips:~# eval "$(docker-machine env linuxtips)"
```

O comando “eval” serve para definir variáveis de ambiente através da saída de um comando, ou seja, as variáveis que visualizamos na saída do “docker-machine env linuxtips”.

Agora que já estamos no ambiente do *host* que criamos, vamos visualizar os *containers* em execução:

```
root@linuxtips:~# docker ps
```

Claro que ainda não temos nenhum *container* em execução; vamos iniciar o nosso primeiro agora:

```
root@linuxtips:~# docker run busybox echo "LINUXTIPS, VAIIII"
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
385e281300cc: Pull complete
a3ed95caeb02: Pull complete
Digest:
sha256:4a887a2326ec9e0fa90cce7b4764b0e627b5d6afcb81a3f73c85dc29cea0004
8
Status: Downloaded newer image for busybox:latest
LINUXTIPS, VAIIII
root@linuxtips:~#
```

Como podemos observar, o *container* foi executado e imprimiu a mensagem “LINUXTIPS, VAIIII”, conforme solicitamos.

Lembre-se de que o *container* foi executado em nosso Docker Host, que criamos através do Docker Machine.

Para verificar o IP do *host* que criamos, faça:

```
root@linuxtips:~# docker-machine ip linuxtips
192.168.99.100
root@linuxtips:~#
```

Para que possamos acessar o nosso *host*, utilizamos o parâmetro “ssh” passando o nome do *host* que queremos acessar:

```
root@linuxtips:~# docker-machine ssh linuxtips
```

Para saber mais detalhes sobre o *host*, podemos utilizar o parâmetro “inspect”:

```
root@linuxtips:~# docker-machine inspect linuxtips
{
  "ConfigVersion": 3,
  "Driver": {
    "VBoxManager": {},
    "IPAddress": "192.168.99.100",
    "SSHUser": "docker",
    "SSHPort": 54031,
    "MachineName": "linuxtips",
    "SwarmMaster": false,
```



```
"SwarmHost": "tcp://0.0.0.0:3376",
"SwarmDiscovery": "",
"StorePath": "/Users/jvitali/.docker/machine",
"CPU": 1,
"Memory": 1024,
"DiskSize": 20000,
"Boot2DockerURL": "",
"Boot2DockerImportVM": "",
"HostOnlyCIDR": "192.168.99.1/24",
"HostOnlyNicType": "82540EM",
"HostOnlyPromiscMode": "deny",
"NoShare": false
},
"DriverName": "virtualbox",
"HostOptions": {
  "Driver": "",
  "Memory": 0,
  "Disk": 0,
  "EngineOptions": {
    "ArbitraryFlags": [],
    "Dns": null,
    "GraphDir": "",
    "Env": [],
    "Ipv6": false,
    "InsecureRegistry": [],
    "Labels": [],
    "LogLevel": "",
    "StorageDriver": "",
    "SelinuxEnabled": false,
    "TlsVerify": true,
    "RegistryMirror": [],
    "InstallURL": "https://get.docker.com"
  },
  "SwarmOptions": {
    "IsSwarm": false,
    "Address": "",
    "Discovery": "",
    "Master": false,
    "Host": "tcp://0.0.0.0:3376",
    "Image": "swarm:latest",
    "Strategy": "spread",
    "Heartbeat": 0,
    "Overcommit": 0,
    "ArbitraryFlags": []
  }
}
```



```
root@linuxtips:~# docker-machine start linuxtips
```

Para removê-lo definitivamente:

```
root@linuxtips:~# docker-machine rm linuxtips  
Successfully removed linuxtips
```

12.2. Agora vamos brincar com o Docker Swarm?

Bom, agora temos uma ferramenta muito interessante e que nos permite construir *clusters* de *containers* de forma nativa e com extrema facilidade, como já é de costume com os produtos criados pelo time do Docker. ;)

Com o Docker Swarm você consegue construir *clusters* de *containers* com características importantes como balanceador de cargas e *failover*.

Para criar um *cluster* com o Docker Swarm, basta indicar quais os *hosts* que ele irá supervisionar e o restante é com ele.

Por exemplo, quando você for criar um novo *container*, ele irá criá-lo no *host* que possuir a menor carga, ou seja, cuidará do balanceamento de carga e garantirá sempre que o *container* será criado no *host* mais disponível no momento.

A estrutura de *cluster* do Docker Swarm é bastante simples e se resume a um *manager* e diversos *workers*. O *manager* é o responsável por orquestrar os *containers* e distribuí-los entre os *hosts workers*. Os *workers* são os que carregam o piano, que hospedam os *containers*.

12.2.1. Criando o nosso *cluster*!

Para criar o nosso *cluster*, vamos rodar o Docker Swarm através de um *container* com a imagem oficial dele do Docker Hub.

Primeiro passo: vamos criar o nosso *cluster* e gerar o *token* que será utilizado como chave para adicionar novos *workers* ao *cluster*.

```
root@linuxtips:~# docker run swarm create  
Unable to find image 'swarm:latest' locally
```

```
Pulling repository docker.io/library/swarm
fd056ae2da24: Pull complete
6b020a24be13: Pull complete
4ae5ea69ee59: Pull complete
da44cf7b75e2: Pull complete
26846c542e9d: Pull complete
dc1bc88e29ba: Pull complete
9e2fe1a1bcde: Pull complete
e4ec6045d936: Pull complete
Status: Downloaded newer image for swarm:latest
b30923039ade73fbdc5beff6b7a65166
```

No código anterior, executamos um *container* com a imagem do “swarm” e pedimos para executar o comando “create”, responsável por criar o *cluster* e gerar o *token* que pode ser visto na última linha da saída.

Como dito anteriormente, precisamos ter um *manager* no *cluster*, que será o responsável por orquestrar o nosso *cluster*. Vamos criá-lo:

```
root@linuxtips:~# docker-machine create -d virtualbox --swarm --swarm-
master --swarm-discovery token://TOKEN_GERADO swarm-
master
Running pre-create checks...
Creating machine...
Waiting for machine to be running, this may take a few
minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Provisioning created instance...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Configuring swarm...
To see how to connect Docker to this machine, run: docker-
machine env swarm-master
```

Temos alguns parâmetros destinados ao Docker Swarm, veja:

- ⇒ **--swarm** – Indica que criaremos um *host* Docker com o perfil “swarm”.
- ⇒ **--swarm-master** – Indica que será o nosso *swarm master*.
- ⇒ **--swarm-discovery** – O “service discovery” que usaremos.

⇒ **token://TOKEN_GERADO** – Usaremos o *token* gerado anteriormente, lembra?

Com isso criamos o nosso *manager* do *cluster*. Lembrando que estamos utilizando o VirtualBox para a criação dos *hosts*, porém você pode utilizar outro *hypervisor* ou ainda alguma nuvem como na AWS, por exemplo.

Agora vamos criar os *swarm nodes*, os *workers* do nosso *cluster*:

```
root@linuxtips:~# docker-machine create -d virtualbox --swarm --swarm-  
discovery token://TOKEN_GERADO swarm-node01
```

```
Running pre-create checks...
```

```
Creating machine...
```

```
Waiting for machine to be running, this may take a few  
minutes...
```

```
Machine is running, waiting for SSH to be available...
```

```
Detecting operating system of created instance...
```

```
Provisioning created instance...
```

```
Copying certs to the local machine directory...
```

```
Copying certs to the remote machine...
```

```
Setting Docker configuration on the remote daemon...
```

```
Configuring swarm...
```

```
To see how to connect Docker to this machine, run: docker-  
machine env swarm-node01
```

```
root@linuxtips:~#
```

```
root@linuxtips:~# docker-machine create -d virtualbox --swarm --swarm-  
discovery token://TOKEN_GERADO swarm-node02
```

```
Running pre-create checks...
```

```
Creating machine...
```

```
Waiting for machine to be running, this may take a few  
minutes...
```

```
Machine is running, waiting for SSH to be available...
```

```
Detecting operating system of created instance...
```

```
Provisioning created instance...
```

```
Copying certs to the local machine directory...
```

```
Copying certs to the remote machine...
```

```
Setting Docker configuration on the remote daemon...
```

```
Configuring swarm...
```

```
To see how to connect Docker to this machine, run: docker-  
machine env swarm-node02
```

```
root@linuxtips:~#
```

Para visualizar os *hosts docker* criados, digite:

```
root@linuxtips:~# docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	SWARM
swarm-master	*	virtualbox	Running		
tcp://192.168.99.100:2376 swarm-master(master)					
swarm-node01	-	virtualbox	Running		
tcp://192.168.99.101:2376 swarm-master					
swarm-node02	-	virtualbox	Running		
tcp://192.168.99.102:2376 swarm-master					

Na coluna “SWARM” temos o nome de quem é o *node master* de cada um dos *hosts docker*; em nosso caso é o “swarm-master”.

Com isso já temos os três *hosts docker* criados, onde um será o *manager* e os demais, *workers*.

Agora, para que tenhamos maiores detalhes de nosso *cluster*, vamos apontar o nosso comando “docker” para trazer informações do *host* “swarm-master”, que é o nosso *manager* do *cluster*.

```
root@linuxtips:~# eval $(docker-machine env --swarm swarm-master)
```

Repare que passamos agora a opção “--swarm”, fundamental para que possamos administrar nosso *cluster swarm* através do *master*.

Agora já “estamos” no *manager* do *cluster* do Docker Swarm. Para verificar as informações referentes ao *cluster*, faça:

```
root@linuxtips:~# docker info
```

Containers: 4
Images: 3
Storage Driver:
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 3

swarm-master: 192.168.99.100:2376

L Status: Healthy

L Containers: 2

L Reserved CPUs: 0 / 1

L Reserved Memory: 0 B / 1.021 GiB

L Labels: executiondriver=native-0.2, kernelversion=4.1.19-boot2docker, operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1);

master : 625117e - Thu Mar 10 22:09:02 UTC 2016, provider=virtualbox, storagedriver=aufs

L Error: (none)

L UpdatedAt: 2016-03-20T22:26:01Z

swarm-node01: 192.168.99.101:2376

L Status: Healthy

L Containers: 1

L Reserved CPUs: 0 / 1

L Reserved Memory: 0 B / 1.021 GiB

L Labels: executiondriver=native-0.2, kernelversion=4.1.19-boot2docker, operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1);

master : 625117e - Thu Mar 10 22:09:02 UTC 2016, provider=virtualbox, storagedriver=aufs

L Error: (none)

L UpdatedAt: 2016-03-20T22:25:40Z

swarm-node02: 192.168.99.102:2376

L Status: Healthy

L Containers: 1

L Reserved CPUs: 0 / 1

L Reserved Memory: 0 B / 1.021 GiB

L Labels: executiondriver=native-0.2, kernelversion=4.1.19-boot2docker, operatingsystem=Boot2Docker 1.10.3 (TCL 6.4.1);

master : 625117e - Thu Mar 10 22:09:02 UTC 2016, provider=virtualbox, storagedriver=aufs

L Error: (none)

L UpdatedAt: 2016-03-20T22:26:03Z

Execution Driver:

Kernel Version: 4.1.19-boot2docker

Operating System: linux

CPUs: 3

Total Memory: 3.064 GiB

Name: a73da4b5e2ed

ID:

Http Proxy:

Https Proxy:

No Proxy:

Como podemos ver, ele nos traz informações sobre a quantidade de *hosts* que compõem o nosso *cluster*, quantos *containers* estão em execução, informações sobre memória, CPU, versões, etc.

A porta utilizada para a comunicação entre os *hosts* e o Docker Swarm é a **2376**.

A partir de agora, quando criarmos *containers*, ele irá distribuí-los entre os *hosts* que fazem parte do *cluster*.

Essa distribuição se baseia na estratégia com a qual você configurou o seu Docker Swarm. Temos hoje disponíveis três estratégias:

- ⇒ **Spread** – Espalha os *containers* entre os *hosts* (com base em CPU, RAM e número de *containers*).
- ⇒ **Binpack** – Concentra mais os *containers* entre os *hosts* (com base em CPU, RAM e número de *containers*).
- ⇒ **Random** – Estratégia randômica. :)

Caso queira alterar a *strategy default*, faça:

```
root@linuxtips:~# docker run -d swarm manage --strategy "binpack"
token://TOKEN_DO_CLUSTER
```

Apenas lembrando que os *containers* criados anteriormente não sofrerão mudança alguma. Continuarão a usar a estratégia que estava definida quando o *container* foi criado. ;)

12.2.2. Docker Swarm Discovery

Falaremos agora sobre uma das peças fundamentais para a implementação de um Swarm em produção. Vamos falar de *Service Discovery* apenas no contexto do Docker Swarm. :)

Já vamos começar desmistificando algo: *Service Discovery* não tem ligação direta com Docker em si; é uma ideia totalmente apartada e usada em outras

coisas, porém cai como uma luva para o projeto, principalmente para a parte de Swarm.

Sendo bem simplista, *Service Discovery* é basicamente o que o nome diz: um modo de fazer a descoberta de “vizinhos” em uma rede. Para isso ele une e gerencia aplicações conhecidas como DHCP e DNS e disponibiliza essas informações via API.

Quando falamos de *Service Discovery*, geralmente estamos falando de duas coisas: *Service Registration*, o processo que vai de fato guardar informações sobre os *hosts*, e o *Service Discovery* em si, que é o processo pelo qual vai ser possível os clientes resgatarem as informações armazenadas.

12.2.3. Mas para que isso serve?

Imagine que você tem um cenário assim: sua aplicação precisa consultar uma API *backend* para disponibilizar informações para o usuário, porém quem responde por essa API na verdade é um grupo de *containers* (ou máquinas) que recebem IP e *hostname* dinamicamente. Como a sua aplicação sabe para onde mandar o *request*? Bom, há duas formas de resolver isso: *client* e *server sides*.

Client side seria assim: sua aplicação faz uma *query* direto no *Service Discovery Backend* que você está usando, e este informa quais máquinas estão disponíveis. Aí basta mandar o *request* direto para a máquina.

Já *server side* seria mais ou menos assim: sua aplicação faz o *request* para um *loadbalance*, e este faz uma *query* no seu *service registry* (outro nome para a mesma coisa), que, por sua vez, manda o IP para o *balancer* e este manda a conexão para um dos servidores.

Docker suporta múltiplos *Service Discovery Backends*; vamos falar um pouco de cada um. No contexto do Docker Swarm, a implementação é bem simples, e todos são implementados da mesma forma.

12.2.3.1. ETCD

ETCD é um simples e poderoso *storage* de chave e valor *open source*, distribuído e escrito em GO, que inicialmente fazia (e ainda faz) parte do projeto CoreOS. Ele gerencia toda a parte de configurações e torna a criação de *clusters* bem mais simples. Ele disponibiliza uma API HTTP que você deve usar para consultar e/ou armazenar informações.

12.2.3.2. Consul

De todos os *storage* chave-valor disponíveis, talvez o Consul seja o mais completo. Com ele você não precisa de nenhum outro software de terceiros para ter um *Service Discovery* implementado com funcionalidades de *healthchecks*, ACLS, etc.

12.2.3.3. Zookeeper

Lembra que começamos esse tópico falando que *service discoveries* em geral eram usados inclusive em outros lugares/projetos *off-Docker*? O Zookeeper é a maior prova disso: está aí desde 2008, inicialmente desenvolvido pelo pessoal do Apache Hadoop para ajudar a manter a consistência entre as suas várias partes.

Estando no mercado há tanto tempo, as maiores vantagens que o Zookeeper tem em relação aos outros são maturidade, robustez e riqueza de *features*. Mas... nem tudo são flores. O Zookeeper é bem complexo, às vezes trabalhoso de ser implementado e/ou de dar manutenção, sem contar que parte dele é escrito em Java, então...

12.2.3.4. Usando um arquivo estático ou uma lista de nodes

Usar um *storage* chave-valor é sempre muito bom e indicado, principalmente para produção, porém não é pré-requisito. Você pode simplesmente usar um arquivo estático para manter esses dados. O lado ruim é ser estático, ou seja, se entrar mais um *node*, você vai precisar alterar o arquivo, visto que um dos pré-

requisitos pra usar um arquivo como *Service Discovery* é ter o arquivo em todas as máquinas do *cluster*. Você vai precisar logar uma a uma e alterá-lo (a não ser que você seja diferente e use aí um NFS da vida pra compartilhar esse arquivo).

12.2.3.5. Docker Hub Service Discovery

Lembra que antes disso tudo a nossa vida era bem mais simples e só tínhamos que especificar esse “token://numeros_aleatorios”?

Esse *token* é gerado pelo *Service Discovery* que fica hospedado no próprio Docker Hub, ou seja, quando você rodou um “swarm create” ele foi até o Docker Hub e criou aquele *hash*, e é com ele que a gente consegue criar e manter o *cluster*. Aí na sua infraestrutura provavelmente você vai ter *firewalls* liberando os tráfegos de entrada e saída, mas isso não será nenhuma novidade se você estiver utilizando o Docker Hub para gerenciar as suas imagens. :)

12.3. E o tal do Docker Compose?

Esta é a ultima peça do ecossistema do Docker.

Com ele você consegue subir aplicações *multicontainers* com apenas um comando. Você cria arquivos, como um *dockerfile*, porém com a diferença de que você pode passar informações sobre todo o seu ambiente e não somente sobre um *container*. Como assim?

12.3.1. Compose File

Com o Docker Compose, você consegue, em um único arquivo, indicar as instruções para a criação de diversos *containers*, tudo o que a aplicação precisar. Com isso você não cria somente um *dockerfile* para criar uma imagem, e, sim, você monta um arquivo YAML com tudo o que a sua aplicação precisa, mesmo que seja em vários *containers*, para criar seu ambiente.

Por exemplo, o arquivo a seguir passa as instruções para criação de um *container* com uma aplicação Python e o outro *container* com o PostgreSQL.

```
db:
  image: postgres
web:
  build: .
  command: python manage.py runserver 0.0.0.0:8000
  volumes:
    - ./code
  ports:
    - 8000:8000
  links:
    - db
```

Com isso, criaremos os dois *containers* executando apenas um comando, o “docker-compose up”.

Antes de qualquer coisa, vamos instalar o Docker Compose. Mais uma vez, como tudo no Docker, este é um processo bastante simples.

12.3.2. Instalando o Docker Compose

```
# wget
https://github.com/docker/compose/releases/download/1.6.2/docker-
compose-Linux-x86_64

# mv docker-compose-Linux-x86_64 /usr/local/bin/docker-
compose
# chmod +x /usr/local/bin/docker-compose
```

Verificando se o Docker Compose foi instalado e sua versão:

```
root@linuxtips:~# docker-compose --version
docker-compose version 1.6.2, build 4d72027
```

Boa! Compose instalado!

12.3.3. Parâmetros do *Compose File*

A seguir listamos algumas opções interessantes que podem ser utilizadas nos arquivos do Docker Compose:

- ⇒ **build** – Indica o caminho do seu *dockerfile*.
- ⇒ **command** – Executa um comando.
- ⇒ **container_name** – Nome para *container*.
- ⇒ **dns** – Indica o DNS *server*.
- ⇒ **dns_search** – Especifica um *search domain*.
- ⇒ **dockerfile** – Especifica um *dockerfile* alternativo.
- ⇒ **env_file** – Especifica um arquivo com variáveis de ambiente.
- ⇒ **environment** – Adiciona variáveis de ambiente.
- ⇒ **expose** – Expõe a porta do *container*.
- ⇒ **external_links** – *Linka containers* que não estão especificados no Docker Compose atual.
- ⇒ **extra_hosts** – Adiciona uma entrada no “/etc/hosts” do *container*.
- ⇒ **image** – Indica uma imagem.
- ⇒ **labels** – Adiciona metadados ao *container*.
- ⇒ **links** – *Linka containers* dentro do mesmo Docker Compose.
- ⇒ **log_driver** – Indica o formato do *log* a ser gerado, por exemplo, *syslog*, *json-file*, etc.
- ⇒ **log_opt** – Indica para onde mandar os *logs*. Pode ser local ou em um *syslog* remoto.
- ⇒ **net** – Modo de uso da rede.
- ⇒ **ports** – Expõe as portas do *container* e do *host*.
- ⇒ **volumes, volume_driver** – Monta volumes no *container*.
- ⇒ **volumes_from** – Monta volumes através de outro *container*.

12.3.4. Vamos fazer o nosso projeto?

Com o Docker Compose já instalado, vamos fazer o nosso primeiro exemplo. Subiremos um Django utilizando o PostgreSQL como DB. Lembrando que,

mesmo com o YAML do Compose, ainda se faz necessária a criação do *dockerfile* com as instruções de criação das imagens envolvidas.

Vamos ao exemplo.

Primeiro o nosso YAML do Compose:

docker-compose.yml

```
root@linuxtips:~# vim docker-compose.yml
db:
  image: postgres
web:
  build: .
  command: python manage.py runserver 0.0.0.0:8000
  volumes:
    - ./code
  ports:
    - 8000:8000
  links:
    - db
```

Agora o *dockerfile* da nossa imagem:

Dockerfile

```
root@linuxtips:~# vim Dockerfile
FROM python:2.7
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
ADD requirements.txt /code/
RUN pip install -r requirements.txt
ADD . /code/
```

O que é necessário para a instalação via *pip* quando o *container* já estiver subindo:

requirements.txt

```
root@linuxtips:~# vim requirements.txt
Django
psycpg2
```

Agora vamos criar o nosso primeiro projeto no Django. Nesse primeiro momento vamos apenas utilizar a seção “web” do nosso “docker-compose.yml”:

```
root@linuxtips:~# docker-compose run web django-admin.py
startproject composeexample .
Pulling db (postgres:latest)...
latest: Pulling from library/postgres
fdd5d7827f33: Already exists
a3ed95caeb02: Pull complete
beb59dc2ad34: Pull complete
f42a5322ef13: Pull complete
f6719ae287c6: Pull complete
0dc08677d778: Pull complete
5f3b03c1dd66: Pull complete
4d4c6707d860: Pull complete
11f8efebbbb9b: Pull complete
edefda034373: Pull complete
5fe4bba523f0: Pull complete
a2d55eea3342: Pull complete
Digest:
sha256:bf1d7cbeb86b76f4c0991061515ebb13c6b57a9588a92f795eaebaede1c3657
b
Status: Downloaded newer image for postgres:latest
Creating root_db_1
Building web
Step 1 : FROM python:2.7
2.7: Pulling from library/python
fdd5d7827f33: Already exists
a3ed95caeb02: Pull complete
0f35d0fe50cc: Pull complete
627b6479c8f7: Pull complete
67c44324f4e3: Pull complete
9ee7e6ec2a05: Pull complete
8772e8de82cd: Pull complete
3fcada98271d: Pull complete
Digest:
sha256:a123fb4fc7203fa58d7afa25afc638954709c4e9887efde28b4cf314deb81d8
0
Status: Downloaded newer image for python:2.7
---> 2ff23583c11e
Step 2 : ENV PYTHONUNBUFFERED 1
---> Running in 39cf98db3246
---> 44932b199d4a
Removing intermediate container 39cf98db3246
```

Step 3 : RUN mkdir /code

---> Running in f8d0ba5b6860

---> 13ea996b14c4

Removing intermediate container f8d0ba5b6860

Step 4 : WORKDIR /code

---> Running in fleef5ff5c93

---> 3fe37473e100

Removing intermediate container fleef5ff5c93

Step 5 : ADD requirements.txt /code/

---> 4117207f329c

Removing intermediate container a6b677c4c7a1

Step 6 : RUN pip install -r requirements.txt

---> Running in f290f8fa010d

Collecting Django (from -r requirements.txt (line 1))

Downloading Django-1.9.4-py2.py3-none-any.whl (6.6MB)

Collecting psycopg2 (from -r requirements.txt (line 2))

Downloading psycopg2-2.6.1.tar.gz (371kB)

Building wheels for collected packages: psycopg2

Running setup.py bdist_wheel for psycopg2: started

Running setup.py bdist_wheel for psycopg2: finished with
status 'done'

Stored in directory:

/root/.cache/pip/wheels/e2/9a/5e/7b620848bbc7cfb9084aafea077be11618c2b
5067bd532f329

Successfully built psycopg2

Installing collected packages: Django, psycopg2

Successfully installed Django-1.9.4 psycopg2-2.6.1

---> 7cf44b49e188

Removing intermediate container f290f8fa010d

Step 7 : ADD . /code/

---> 579234c814e7

Removing intermediate container 2559c0443084

Successfully built 579234c814e7

root@linuxtips:~#

Agora já temos o projeto Django chamado “composeexample”, criado no mesmo diretório em que estávamos, conforme notamos a seguir:

root@linuxtips:~# ls

composeexample docker-compose.yml Dockerfile manage.py
requirements.txt

Agora vamos ajustar apenas a configuração para conexão ao banco no arquivo “settings.py” que está dentro do diretório “composeexample”.

Vamos alterar somente a seção “DATABASES”:

```
root@linuxtips:~# cd composeexample
root@linuxtips:~# vim settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'HOST': 'db',
        'PORT': 5432,
    }
}
```

Agora que já ajustamos as informações referentes ao nosso banco, já podemos subir o nosso Compose. ;)

Para isso, basta fazer, no diretório onde se encontra o arquivo “docker-compose.yml”:

```
root@linuxtips:~# docker-compose up
Starting livro_db_1
Starting livro_web_1
Attaching to livro_db_1, livro_web_1
db_1 | LOG: database system was shut down at 2016-03-20 23:09:33 UTC
db_1 | LOG: MultiXact member wraparound protections are now enabled
db_1 | LOG: database system is ready to accept connections
db_1 | LOG: autovacuum launcher started
web_1 | Performing system checks...
web_1 |
web_1 | System check identified no issues (0 silenced).
web_1 |
web_1 | You have unapplied migrations; your app may not work properly
web_1 | until they are applied.
web_1 | Run 'python manage.py migrate' to apply them.
web_1 | March 20, 2016 - 23:09:36
web_1 | Django version 1.9.4, using settings
web_1 | 'composeexample.settings'
web_1 | Starting development server at http://0.0.0.0:8000/
web_1 | Quit the server with CONTROL-C.
```

Quando você utilizar o comando “docker-compose up”, ele iniciará o projeto subindo os *containers* que estão descritos no arquivo “docker-compose.yml”, conforme desejamos. :D

Você consegue visualizar os *logs* durante a subida do projeto do Docker Compose, porém, caso queira que o projeto suba em segundo plano, como um *daemon*, basta utilizar o parâmetro “-d”, conforme segue:

```
root@linuxtips:~# docker-compose up -d
Starting livro_db_1
Starting livro_web_1
```

Perfeito! Como podemos ver na saída do comando, nosso *web server* está em pé e respondendo na porta 8000!

Para validar, vamos executar um “curl” batendo na porta 8000 do nosso *host*:

```
root@linuxtips:~# curl localhost:8000
<!DOCTYPE html>
<html lang="en"><head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <meta name="robots" content="NONE,NOARCHIVE"><title>Welcome to
Django</title>
  <style type="text/css">
    html * { padding:0; margin:0; }
    body * { padding:10px 20px; }
    body * * { padding:0; }
    body { font:small sans-serif; }
    body>div { border-bottom:1px solid #ddd; }
    h1 { font-weight:normal; }
    h2 { margin-bottom:.8em; }
    h2 span { font-size:80%; color:#666; font-weight:normal; }
    h3 { margin:1em 0 .5em 0; }
    h4 { margin:0 0 .5em 0; font-weight: normal; }
    table { border:1px solid #ccc; border-collapse: collapse;
width:100%; background:white; }
    tbody td, tbody th { vertical-align:top; padding:2px 3px; }
    thead th {
      padding:1px 6px 1px 3px; background:#fefefe; text-
align:left;
      font-weight:normal; font-size:11px; border:1px solid #ddd;
    }
  </style>
  <div>
    <h1>Welcome to Django</h1>
    <h2>Very nice!</h2>
    <table>
      <thead>
        <tr>
          <th>Model</th>
          <th>Count</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Author</td>
          <td>1</td>
        </tr>
        <tr>
          <td>Book</td>
          <td>2</td>
        </tr>
      </tbody>
    </table>
  </div>
</html>
```

```

tbody th { width:12em; text-align:right; color:#666;
padding-right:.5em; }
#summary { background: #e0ebff; }
#summary h2 { font-weight: normal; color: #666; }
#explanation { background:#eee; }
#instructions { background:#f6f6f6; }
#summary table { border:none; background:transparent; }
</style>
</head>

<body>
<div id="summary">
  <h1>It worked!</h1>
  <h2>Congratulations on your first Django-powered page.</h2>
</div>

<div id="instructions">
  <p>
    Of course, you haven't actually done any work yet. Next, start
    your first app by running <code>python manage.py startapp
    [app_label]</code>.
  </p>
</div>

<div id="explanation">
  <p>
    You're seeing this message because you have <code>DEBUG = True</code>
    in your Django settings file and you haven't
    configured any URLs. Get to work!
  </p>
</div>
</body></html>

```

Aeeeeee!!! \o/

Congratulations on your first Django-powered page!!!

Com o comando “docker-compose ps” você consegue visualizar nosso projeto:

```
root@linuxtips:~# docker-compose ps
```

```
    Name    Command    State
```

```
Ports
```

```
-----
```

```

-----
livro_db_1    /docker-entrypoint.sh postgres Up
5432/tcp

livro_web_1   python manage.py runserver ... Up
0.0.0.0:8000->8000/tcp

```

Aumentar o número de *containers* DB é bastante simples: basta utilizar o comando “docker-compose scale” conforme segue:

```
root@linuxtips:~# docker-compose ps
```

```

      Name Command State
      Ports
-----
-----
livro_db_1    /docker-entrypoint.sh postgres Up 5432/tcp
livro_db_2    /docker-entrypoint.sh postgres Up 5432/tcp
livro_web_1   python manage.py runserver ... Up
0.0.0.0:8000->8000/tcp

```

Fácil, não? :D

Caso queira parar todos os *containers* desse *compose*, basta fazer conforme segue:

```
root@linuxtips:~# docker-compose stop
```

```

Stopping livro_db_2 ... done
Stopping livro_web_1 ... done
Stopping livro_db_1 ... done

```

Caso queira iniciar novamente o *compose* dos *containers*:

```
root@linuxtips:~# docker-compose start
```

```

Starting livro_db_2
Starting livro_db_1
Starting livro_web_1

```

Caso queria verificar os *logs*:

```
root@linuxtips:~# docker-compose logs
```

12.4. E já acabou? :(

Esperamos que você tenha curtido viajar conosco durante o seu aprendizado sobre *containers* e principalmente sobre o ecossistema do Docker, que é sensacional!

Não pare de aprender mais sobre Docker! Continue acompanhando o Canal LinuxTips no <http://youtube.com/linuxtips> e fique ligado no site do Docker, pois sempre tem novidades e ótima documentação!

Junte-se a nós no <http://facebook.com/linuxtipsbr> para que possa acompanhar e tirar dúvidas que possam ter surgido durante seus estudos!

#VAIIII